SANDIA REPORT

SAND2011-6603
Unlimited Release
Printed September 2011

Solving the Software Protection Problem with Intrinsic Personal Physical Unclonable Functions

Rishab Nithyanand, Radu Sion, John Solis

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2011-6603
Unlimited Release
Printed September 2011

Solving the Software Protection Problem with Intrinsic Personal Physical Unclonable Functions

Rishab Nithyanand
Stony Brook University
Stony Brook, NY, USA
rnithyanand@cs.stonybrook.edu

Radu Sion
Stony Brook University
Stony Brook, NY, USA
sion@cs.stonybrook.edu

John H. Solis
Sandia National Laboratories
Livermore, CA, USA
jhsolis@sandia.gov

Abstract

Physical Unclonable Functions (PUFs) or Physical One Way Functions (P-OWFs) are physical systems whose responses to input stimuli (i.e., challenges) are easy to measure (within reasonable error bounds) but hard to clone. The unclonability property comes from the accepted hardness of replicating the multitude of characteristics introduced during the manufacturing process. This makes PUFs useful for solving problems such as device authentication, software protection, licensing, and certified execution. In this paper, we focus on the effectiveness of PUFs for software protection in offline settings.

We first argue that traditional (black-box) PUFs are not useful for protecting software in settings where communication with a vendor's server or third party network device is infeasible or impossible. Instead, we argue that *Intrinsic PUFs* are needed to solve the above mentioned problems because they are intrinsically involved in processing the information that is to be protected. Finally, we describe how sources of randomness in any computing device can be used for creating intrinsic-personal-PUFs (IP-PUF) and present experimental results in using standard off-the-shelf computers as IP-PUFs.

Acknowledgments

We would like to thank John Floren, Ron Minnich, and Don Rudish of Sandia National Laboratories for providing us with the identical hardware used to conduct the preliminary tests and experiments.

Part of this work was funded by the Laboratory Directed Research and Development (LDRD) program at Sandia National Laboratories. Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Contents

1 Introduction and Motivation

Physical Unclonable Functions (PUFs) or Physical One Way Functions (P-OWFs) are physical systems whose responses to input stimuli (i.e., challenges) are easy to measure (within reasonable error bounds) but hard to clone. The unclonability property comes from the accepted hardness of replicating the multitude of uncontrollable manufacturing characteristics.

In a nutshell, PUFs rely on hiding *secrets* in circuit characteristics rather than in digitized form. On different input stimuli (challenges) a PUF circuit exposes certain measurable and persistent characteristics (responses). Several varieties of PUFs have been proposed since being introduced by Pappu in [11] and range from optical PUFs that analyze the speckle pattern resulting from shining a laser beam on a transparent object to silicon timing PUFs.

While PUF technology was initially envisaged as a new tool for simple device identification and authentication, the attractiveness of the unclonability property has greatly broadened the scope of its possible applications. Current and emerging applications range from software protection and licensing to hardware tamper proofing and certified execution. The common purpose of PUF technologies in these applications is efficient hardware identification and authentication via circuit measurements. Out of these applications, the most interesting problem with the largest potential impact is the *software protection problem*.

Instead of building functionality directly into hardware, it is often more cost-effective to produce generic computing hardware that provides its functionality via software. Software solutions have shorter development, testing, and deployment life cycles, and as a result, have become established in a wide variety of products and markets. Yet despite the benefits, software makers constantly struggle with the illegal duplication and reverse engineering of their software and actively seek protection mechanisms. The goal of any software protection scheme is to develop a mechanism such that any tampering to provide illegal monetary benefits (i.e., via piracy and illegal execution) is infeasible.

Unfortunately, current schemes do an insufficient job at protecting software – piracy costs the software industry a colossal loss of $51.4 billion annually [13]. Protecting software in offline scenarios is extremely challenging because a malicious host has complete control, access, and visibility over any piece of software it executes. Well designed schemes that incorporate trusted devices (i.e., hardware or servers) can prevent a malicious host from having complete control over the software. However, requiring additional hardware or online access is not always feasible in all scenarios. This leads us to ask whether it is even possible to use PUFs in a software protection scheme.

In the "real world", a major security issue with using PUFs for software protection and licensing is dealing with PUF replay and virtualization attacks [1] – also referred to as OORE (Observe Once, Run Everywhere) attacks. Software protection schemes can deal with these attacks by using trusted hardware (e.g., ORAM) or trusted servers. However, these methods are untenable in the (hostile) offline scenario involving PUFs [1]. All protection mechanisms can be bypassed by running the software in a virtual machine with a virtual PUF primed to behave as the legitimate PUF

would after observing just a single run in the legitimate environment.

In this paper, we aim to solve the software protection problem with the aid of PUF technology. Software protection is a discipline that falls between the gaps of theory, software and hardware engineering. As a result, there are tremendous research opportunities with the potential to develop into practical solutions.

1.1 Contributions

Our primary contribution is an investigation into the feasibility of using PUFs technology to solve the software protection problem. We show that traditional (black-box) PUFs are not useful for protecting software in offline settings where communication with a vendor's server or third party network device is infeasible or impossible. Instead, we argue that *Intrinsic PUFs* are needed to solve this and other problems, such as, software licensing and certified execution, because they are intrinsically involved in processing the information that is to be protected. Finally, we describe how sources of randomness in any computing device can be used for creating intrinsic-personal-PUFs (IP-PUF) and present the results of our experiments to use standard off-the-shelf computers as IP-PUFs.

1.2 Related Work

The first work geared towards the anti-piracy and software protection problem was in 1980 by Kent [9]. Kent suggested the use of tamper resistant trusted hardware and encrypted programs. He was also the first to differentiate the trusted host problem from the trusted code problem. In 1985, Gosler [7] proposed the use of dongles and magnetic signatures in floppy drives along with several anti-debugging techniques to prevent software analysis and copying. Unfortunately, these early works are vulnerable to an OORE attack.

In 1993, Cohen [4] proposed a solution using software diversity and code obfuscation as a software protection mechanism. Cohen's methods were based on simple code transformation and obfuscation techniques. Additional transformation and obfuscation techniques were later proposed by Collberg *et al*[5] and Wang [14]. Finally, Goldreich and Ostrovsky provided the first theoretical analysis and foundation to the software protection problem in 1990 and later again in 1996 [6]. They used schemes to hide/obfuscate data access patterns in conjunction with trusted hardware to prevent illegal software replication. More recently, Boaz Barak *et al*[2] completed a theoretical analysis of software obfuscation techniques. Their contribution was an interesting negative result that, in essence, proved that there exists a family of programs that are non- learnable and yet, un-obfuscatable (by any code obfuscator). Therefore, this implies (in its most extreme interpretation) that there does not exist a provably secure obfuscation algorithm that works on every program. In a new approach to the problem, Chang and Atallah [3] proposed a scheme that prevented software tampering using a set of inter-connected (code) guards programmed to perform code verification and repairs.

After the advent of Physical Unclonable Functions, there have been several proposals to use them for software protection. Most notably, Guajardo *et al.* in 2007 [8], proposed an FPGA based IP protection scheme that relied on SRAM PUFs. However, SRAM PUFs are not ideal due to the possibility an exhaustive read out attack. Atallah *et al.* provided the inspiration for this work with [1]. They proposed inter-twining software functionality with the PUF. However, there were several drawbacks to their approach: (1) it requires trusted hardware to remotely initialize the protection scheme and (2) can only protect software with algebraic group functionality.

2 Preliminaries

2.1 PUF Definition

Although the formal definition of a PUF has been under debate recently [12], we use the following definition throughout this paper:

Definition 1. *A Physical Unclonable Function is a physical system that possesses the following properties:*

- *(Persistent and Unpredictable) The response (R_i) to some challenge (C_i) is random, yet persistent over multiple observations.*
- *(Unclonable) Given a PUF (P'), it is infeasible for an adversary to build another system (P'') – real or virtual – that provides the same responses to every possible challenge.*
- *(Tamper Evident) Invasive attacks on the PUF essentially destroy them and render them ineffective.*

It is important to note that a randomness property is not explicitly required since the notion of unclonability supersedes the notion of randomness.

2.2 An Impossibility Conjecture

Our first task is to show why traditional (black-box) PUFs are not useful in protecting software. The following informal argument intuitively shows why it is impossible to build a provably secure software protection scheme without using trusted hardware for secure storage and/or processing. The formal theoretical results are fully presented in [10]. Note that the following arguments also apply to the related areas of software obfuscation, whitebox cryptography, and software watermarking.

Conjecture 1. *There cannot exist a provably secure software protection scheme in an offline setting without trusted hardware.*

Reasoning. It is clear that randomness must be involved during the process of selecting challenges. This leads us to set up our program P' as a probabilistic Turing machine PTM. As with other probabilistic Turing machines, PTM behaves like an ordinary deterministic Turing machine except for the following:

- Multiple state transitions may exist for certain entries of the state transition function.
- Transitions are made based on probabilities determined by a random tape R consisting of a binary string of random bits.

We say that $x \in L(PTM_y)$ if $PTM(x,y)$ halts and accepts. Here, y represents the bits on the random tape. For our machine PTM, the input tape is write enabled and consists of bits that determine the computation path and responses to challenges issued by the transition function. The transition function, at each challenge stage, may select one of a large finite number of challenges based on the string of bits y in the random tape R. At the verify response stage, the transition function may make a state transition based on the response received to the issued challenge. Any input x that requests a valid computation and contains correct responses to all challenges issued by the transition function will result in a halt and accept state.

A fundamental requirement for all probabilistic Turing machines is that the random tape R is read-only (i.e., it is not write enabled). However, in the purely offline setting where there exists no trusted hardware, it is impossible to enforce this requirement – every tape of the Turing machine PTM is write-enabled. Since the random tape is write enabled, an adversary may force the bits y on it to enforce a certain set of challenges on every iteration of PTM – resulting in a deterministic Turing machine $PTM(x)$ rather than the probabilistic machine $PTM(x,y)$. This permits an adversary to launch an OORE attack as described earlier.

3 Rethinking the Software Protection Problem

As discussed in Section 2.2, it is impossible to achieve a provably secure software protection mechanism without using trusted hardware or an online server. Our goal is to find a feasible *offline* solution that does not require additional trusted hardware. To this end, we re-analyze the software protection problem and explain why traditional (i.e., black-box) PUFs and traditional models of computing (i.e., Turing machines and RAM) fail to provide headway towards a solution. In the next section, we describe a new type of PUF – the Intrinsic Personal PUF, that solves the IP-Protection problem.

3.1 Failure of Traditional PUFs

The main reason why traditional PUFs fail is the impossibility of supplying random challenges to the PUF from a deterministic program. Further, the PUF is a peripheral device that executes with the program via some bus where, in a hostile environment, any information transferred over the bus is known and monitored by the adversary. This allows an adversary to easily replicate/virtualize the PUF by replaying the recorded data. This makes black-box PUFs vulnerable to replay/virtualization attacks and renders them unusable against OORE attacks.

This leads us to recognize the need for a PUF which is intrinsically involved in the actual computation performed by the program, e.g., a processor that exhibits certain timing characteristics. We call such PUFs *intrinsic* and *personal*: intrinsic because they are inherently involved during software execution and personal because every computing device possesses such a PUF.

Intrinsic Personal PUFs (IP-PUFs) are PUFs that are intrinsically and continuously involved in the computation of the program to be protected.

3.2 Failure of Traditional Computing Models

Traditional Turing machine and RAM computing models fail to model the software protection problem and their use with IP-PUFs because intrinsic features and randomness introduced via defects in the manufacturing process cannot be sufficiently modeled. Any attempts to find a purely PUF based solution will fail since a provably secure solution does not exist. Instead, the offline protection problem should rely on a systems oriented approach rather than a theoretical one.

4 Intrinsic Personal Physical Unclonable Functions (IP-PUFs)

4.1 Preliminaries

Silicon timing PUFS, as the name suggests, analyze timing behaviors of a circuit to determine an appropriate response to a given input. However, the very existence of this class of PUFs raises many interesting questions: Can a personal computer (which is itself a large silicon/crystal circuit) be used as an intrinsic PUF device? If so, which system characteristics are most promising as PUF characteristics? But perhaps most importantly: can any personal electronic device be used to build truly secure PUF based protocols?

We set out to investigate the questions posed above and answer the first question in the affirmative by claiming that it is possible (within reasonable error bounds) to use regular computers as *intrinsic (i.e., non-black-box) crystal/silicon based timing PUFs* (where the challenge is an instruction, and execution time is the *response*). We argue that this behavior is sufficient for preventing the replay / virtualization attacks and enables implicit hardware identification without requiring peripheral PUF devices.

We investigate several system characteristics and present initial experimental results answering the following questions:

- **Intra-Architecture Variations:** Are there measurable timing differences across systems with identical architectures and specifications and with all components belonging to the same family?
- **Challenge-Response Variation:** Are there measurable timing differences for different challenges (i.e., instructions) with different inputs on the same machine?
- **Inter-Architecture Variations:** Are there measurable timing differences across systems with different architectures and similar specifications, with parts not belonging to the same family, etc.?

Finally, we briefly explain how these can provide better software protection and continuous authentication.

4.2 IP-PUFs: Sources of Unpredictability and Randomness

Given a stable environment and operating conditions (i.e., controllable and static temperature, pressure, voltage supply, etc.), the following are the most interesting and common sources of unpredictability and unclonability in personal devices (e.g., computers, mobile phones, game consoles, appliances) suitable for use as timing PUFs:

- **System Clock Skews and Bus Delays:** The primary clock generators available on most computing units that affect timing are: (1) the front-side bus (FSB) clock on the northbridge controller, (2) the programmable interrupt timer (PIT) clock on the southbridge controller, and (3) the PCI-e clock. Additional clocks, such as, the GPU core and memory clocks, may also be used for measuring timing characteristics for device identification. Clocks are interesting for several reasons. Given two clocks (from the same or different clock synthesizers) labeled with identical frequencies, it is unlikely that both oscillate at identical rates. This is due to several factors, such as crystal cut, impurities, age, and clock synthesizer circuit variants.

  Here we describe the effects of variations of system clocks (i.e., FSB clock, PIT clock, PCI-e clock, etc.) on various components of a computing device.

  - **Central Processing Unit:** An interesting consequence can be observed when comparing two different FSB clock sources: the actual time (in *picoseconds*) for instruction execution on two identical (in specification) CPUs is different even though the required number of clock cycles is identical. This can easily be confirmed by observing the varying *bogomips*[2] values for processors from the same batch and family.

    Further, differences in the oscillating frequency of the timer interrupt clock (PIT clock) cause different definitions of a time quantum (i.e., for process schedulers performing round-robin scheduling) across multiple identical systems. The combination of the PIT and FSB clocks results in a different number of clock cycles, and therefore actual instructions, being executed during *one time quantum*. These small differences can be used to identifying the CPU and the clock generating unit of any computing device.
  - **Synchronous Dynamic RAM (SDRAM):** SDRAM operates in a synchronized fashion with the FSB clock to respond to all control signals. The exact operating frequency of the memory interface is related to the FSB by a (configurable) gearing ratio. Differences in memory read/write operation latency across identical rated chips can be computed as: $cycles \times \delta(Clock_{FSB}) \times GearingRatio$. This delay can be used to identify the Memory Control Hub (MCH) and the SDRAM units of a computing device.
  - **Graphical Processing Unit and Video RAM:** The GPU is driven by a core clock and a memory clock. The effect of two GPUs, identical in specification, are the same as those described with the CPU. Identical instructions take slightly different times to execute on identical (in specification) hardware. These delays can identify the GPU and the PCIe controller used in a computing device.
  - **PCI Express Bus:** Peripheral devices such as audio cards, network cards, modems, tuner cards, etc., are connected to the northbridge controller (memory control hub) via the PCI Express bus. The PCI Express bus is driven by a 100 MHz clock (asynchronous to the FSB clock). This 100 MHz clock permits data rates of up to 2.5 Gbps in single (x1) link PCI Express (most peripheral devices employ a 25x clock multiplier). However, as mentioned before, the difference in the actual clock speed means that it is unlikely that any two PCI Express devices with same specifications and interfaces will

[2] *Bogomips* is an inaccurate and low-precision measure of CPU speed. It is measured using a busy loop while booting and is accessible from `/proc/cpuinfo` on Linux systems.

actually be able to achieve this exact transfer rate. The difference in data throughput across identically rated chips is $cycles \times \delta(Clock_{PCIe}) \times 25 \times (linksize)$. These delays are useful in identifying PCIe peripherals and the PCIe controller of a computing device.

- **System Management Bus:** The SMBus is a (southbridge connected) low bandwidth communication channel used primarily for communication with sensors and power sources. In general, multiple masters or slaves may be connected to the SMBus. However, only one master may control the bus at any point in time. The SMBus operating frequency is defined to be 100KHz, but due to variations in the clock sources, are unlikely to operate at this exact rate. As a result, it is possible to identify an SMBus controller based on its clock variations. Further, this frequency combined with the time taken for a slave to issue an *SMBALERT* may also be used to identify connected sensors (i.e., slaves).
- **PCI Bus:** The PCI bus is driven by a 33Mhz or 66MHz clock from the southbridge controller. Older networking cards, video cards, audio controllers, and tuners are usually connected to the PCI Bus. The typical transfer rate for a 64 bit wide PCI bus is 266MB/s (or 533 MB/s, for a 66MHz clock). However, due to varying frequencies of generated clocks, this transfer rate is unlikely to be identical across multiple systems. Therefore, the difference in data throughput across identical rated chips is $cycles \times \delta(Clock_{PCI}) \times PCIMultiplier \times (linksize)$. These delays are useful in identifying PCI buses, controllers, and peripheral devices of a computing device.
- **Universal Serial Bus:** The USB (v2.0) is driven by a 48MHz clock from the southbridge controller with data transfer rates of 480 Mb/s. However, due to varying frequencies of the generated USB clock, this transfer rate is unlikely to be identical across multiple systems. Therefore, the difference in data throughput across identical rated chips is $cycles \times \delta(Clock_{USB}) \times 10$. These delays are useful in identifying the USB controller, bus, and peripheral devices of a computing device.
- **SATA Bus:** The SATA bus is driven by a 100MHz clock connected to the southbridge controller with a data transfer rate for SATA 3 devices at 6Gb/s. However, due to varying frequencies of the generated SATA clock, this transfer rate is unlikely to be identical across identical (in specification) hardware. The difference in data throughput for identical specification systems is given by $cycles \times \delta(Clock_{SATA}) \times 60$. These delays are useful in identifying the SATA controller, bus, and SATA devices of a computing system. However, it is important to note that these delays are measurable only when the bottleneck of the communication/transfer operation is the buffer to processor latency and not the device to buffer latency (as is the case with traditional hard disk drives).

- **Mass Memory Storage Devices:** Mass memory storage devices are synchronized with the FSB clock. However, they do not typically offer guaranteed service latencies for memory read/write operations (as opposed to main memory and cache memory). We now observe how this characteristic can be used to extract unique behaviors for device identification (and the host computing system).

  - **Magnetic HDD:** The main contributors to HDD latency/disk-to-buffer transfer rate are

rotational latency and seek time. Rotational latency is dictated by the rotational speed of the magnetic head (in *rpm*) and is the delay taken by the drive head to reach the appropriate track. Analysis of performance specifications indicate a +/- 0.1 - 1 % error rate in the rotational speeds from labeled specifications. This contributes to the differences in rotational latency (and disk-to-buffer latency) across drives with identical specifications. Track-to-track latency (i.e., seek time), however, does not reflect any error rate. We argue that the differences in rotational latency (i.e., disk-to-buffer latency) allow us to identify a hard-drive within reasonable error-rates. This can be extended to the entire computing device when used in conjunction with the previously mentioned methods and devices.

  - **Solid State Drives:** Analysis of product specifications and benchmarking data indicate that the read/write latency of identical solid-state drives vary on the order of tens of nanoseconds for identical hardware. These variations allow us to identify a SSD within reasonable error-rates (and the entire computing device when used in conjunction with the previously mentioned methods and devices).

- **Asynchronous Dynamic RAM:** Although ADRAM is less common in modern computing devices, their presence on a system allows for increased entropy and variability when used as a PUF.

  ADRAM operates in an asynchronous manner and responds to control signals as soon as data is available – regardless of the FSB clock cycle. These times are different for ADRAM chips with the same specifications and this variation can again be harnessed for device identification purposes. In particular, the time taken for an instruction requiring access to data in ADRAM is different in the order of picoseconds even across identical hardware.

- **Overclocked Stability:** Another method of identifying computing devices and their individual components to achieve more entropy is to measure behavior and stability patterns when overclocked. An overclocked device/component has a random (yet persistent) pattern of logic gate misfiring and failure that is potentially observable. This behavior is otherwise hidden under stable conditions.

  Components that may be identified by measuring overclocked stability patterns include (but are not limited to): [Northbridge components] CPUs, RAM, GPUs, and any PCI-e component; [Southbridge components] any PCI component.

4.3 Preliminary Experiments

We conducted a series of experiments to assess our approach using a set of identical off-the-shelf computer systems and aim to understand the degree of timing variations for a set of instructions. To eliminate as many variables as possible, we conducted our experiments under controlled operating environments, on five identical machines with components (processors, PSU's, and RAM) that were from the same batch and family. The system specifications were: 2.80 GHz Quad-core Intel Core i7 930 processors (Bloomfield architecture), 2MB L3 cache/core, 256KB L2 cache/core, and 12GB DDR3 RAM, completely diskless.

In this preliminary experiment, we only seek to identify computers by studying the variations in number of instruction cycles per time quantum. All instructions and data were loaded into the CPU cache. Delays due to memory, buses, and I/O were not studied.

Experimental Setup

Certain precautions were taken to ensure valid and consistent results: First, several BIOS/Kernel changes were made to disable dynamic voltage and CPU frequency stepping. Next, we had to ensure that our benchmarking application was free from interrupts and the vagaries of the OS process scheduler by blocking all process signals and interrupts. Our application was swapped out every 10 ms and the number of instruction cycles completed was obtained by reading the TSC register(Time Stamp Counter).

Our program performed 10 million simple mathematical operations using the same inputs and was only allowed to execute on one core for each experiment, i.e., hard affinity to a specific CPU was set. We argue that the timing inaccuracies of the TSC register are not harmful to our measurements since the values are used to classify identical systems and not for computations.

Experiments were repeated 1000 times over multiple sessions to gather training data for a classifier. Finally, a test sample was collected and input to the classifier to identify systems.

Experiment Results and Analysis

Recall that we aimed to answer the following questions: (1) Intra-Architecture Variations: Are there measurable timing differences across systems with identical architectures and specifications and with all components belonging to the same family? (2) Challenge-Response Variation: Are there measurable timing differences for different challenges (i.e., instructions) with different inputs on the same machine? (3) Inter-Architecture Variations: Are there measurable timing differences across systems with different architectures and similar specifications, with parts not belonging to the same family?

Intra-Architecture Variation: Our results are illustrated in Fig. 1. After training, our classifier had a 100% true positive rate with a 62% false positive rate.

Challenge-Response Variation: Certain instructions require more clock cycles or computations from different components of the computing device (e.g., floating point operations inherit the timing characteristics and delays of the floating point unit). We confirmed through our experiments that even the same arithmetic operations, when using varying inputs, had slightly different execution times on the same system. In essence, this allows us to use any available mathematical or logical operation as a challenge to the computing device.

Inter-Architecture Variation: As part of a preliminary study, we experimented with various machines having different architectures (i.e., Bloomfield vs. Westmere) and slightly different spec-

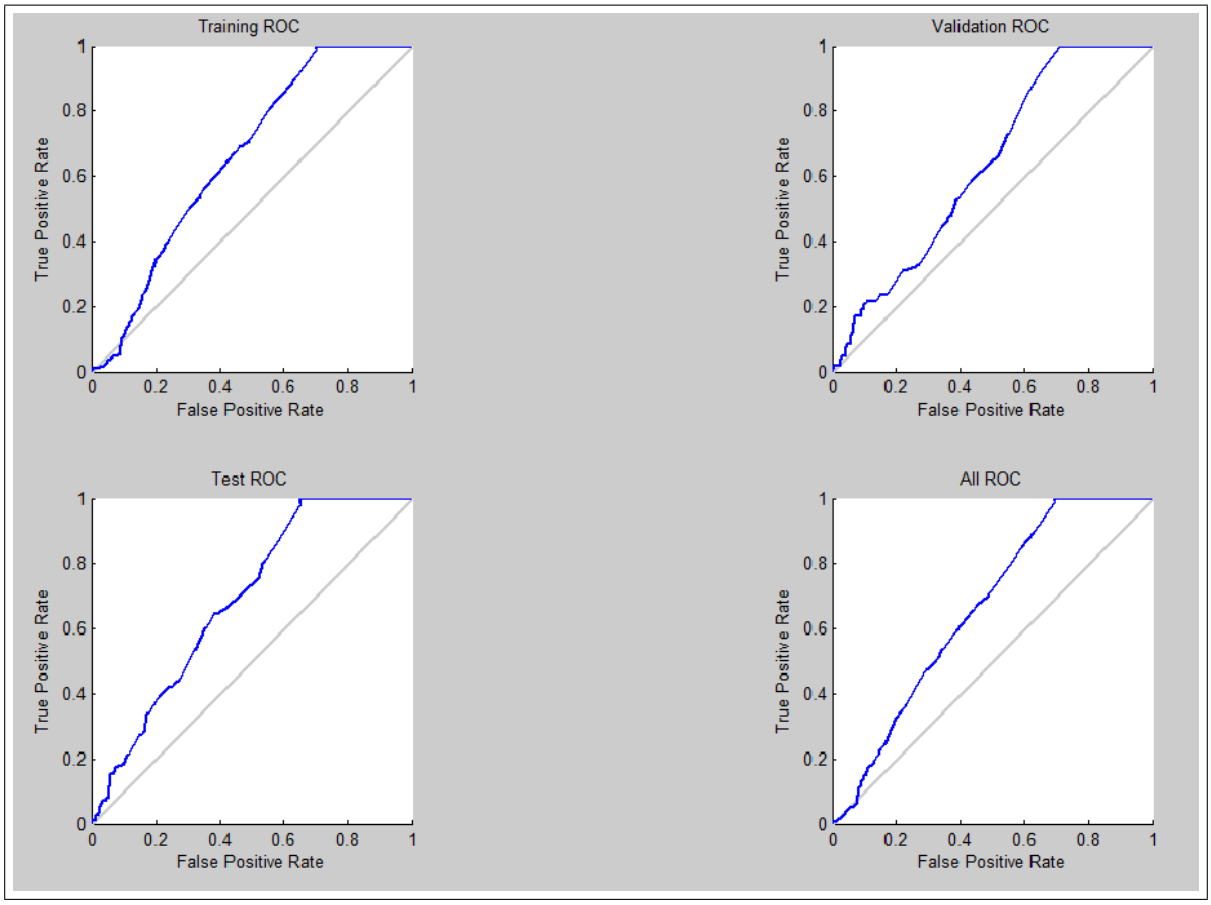

Figure 1. Intra Architecture Classifier Results. True Positives vs. False Positives (Training, Validation, Test, and Combined Results).

ifications. We were able to successfully perform classification with 100% accuracy (i.e., with no false positives or false negatives).

4.4 IP-PUFs and the Software Protection Problem

The natural question that arises at this point is: how can we actually use IP-PUFs to protect software and ensure that it is not capable of being executed on non-licensed devices.

Our approach is to use IP-PUFs that are not directly measured in the traditional challenge-response sense. Rather, we analyze the timing and error characteristics of an IP-PUF to build *error driven* software that expects these characteristics/errors to occur in the hardware. The absence of these errors should lead the software to execute incorrectly. One example of such a scheme is to insert artificial race conditions (using non-semaphored threads) in a program. These threads

are resolved correctly (i.e., the correct order of variable clobbering is maintained) only when the timing characteristics of the (expected) IP-PUF are observed. On an incorrect device, the threads will eventually clobber shared variables in an incorrect order, therefore leading the program to crash or return incorrect values.

Such methods may not be provably secure against the OORE adversary, however, they accomplish several interesting feats: (1) They raise the bar for an adversary to defeat the protection mechanism since race conditions are extremely difficult to debug, and (2) The dependence on the timing characteristics of the computing device (i.e., using the device as a time based IP-PUF) ensures that typical adversary tools such as debuggers and virtual machines are no longer usable (since they change the timing characteristics analyzed programs).

5 Conclusions and Future Work

PUFs have been envisioned as applicable to many practical problems such as hardware authentication, certified execution, and most notably software protection. However, every current approach that attempts to use PUFs for offline hardware authentication and software protection is vulnerable to virtualization attacks.

Current offline device authentication and software protection schemes rely on discrete (i.e., static) authentication schemes. These provide no differentiation between an authentic device and a virtual device that replicates the *useful* part of the device (i.e., the parts that are actually challenged). Using IP-PUFs as described here, reduce these attacks significantly by continuously authenticating the device implicitly and transparently. Further, this authentication method is useful for software protection by intertwining software with a specific computing device (e.g., by inserting race conditions that resolve correctly only on the correct device). This approach makes it difficult for any adversary to unhook software functionality from the PUF since many traditional debugging tools are now useless.

To summarize our contributions, we first showed why traditional PUFs are not useful in solving the software protection problem in offline scenarios. Instead, we argue that intrinsic-personal PUFs are needed since they are intrinsically involved in processing the information that is to be protected. We presented the preliminary results of our experiments to use off-the-shelf computers as IP-PUFs and show how with timing characteristics alone, we achieved a 60% success identification rate across identical hardware. We conclude that IP-PUFs are already widespread, easily available, and easy to measure (via benchmarking suites) without the need for additional hardware. We also show how IP-PUFs can be used as a basis for offline continuous device authentication and software protection. Further, IP-PUFs raise the bar for an attacker by negating the usefulness of virtual machines and debugging tools.

As part of our future work, we plan to conduct further experiments on additional computers (up to 10-20 nodes) with identical components to test the validity of our hypothesis that off-the-shelf computing devices can be successfully identified and authenticated using their timing characteristics. We also plan on building software that is intertwined to a particular computer by its timing characteristics and by constructing race conditions as described above.

References

[1] Mikhail J. Atallah, Eric D. Bryant, John T. Korb, and John R. Rice. Binding software to specific native hardware in a VM environment: the puf challenge and opportunity. In *VMSec '08: Proceedings of the 1st ACM workshop on Virtual machine security*, pages 45–48, New York, NY, USA, 2008. ACM.

[2] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-44647-8_1.

[3] Hoi Chang and Mikhail J. Atallah. Protecting software code by guards. In *Digital Rights Management Workshop*, pages 160–175, 2001.

[4] Frederick B. Cohen. Operating system protection through program evolution. *Comput. Secur.*, 12:565–584, October 1993.

[5] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, July 1997.

[6] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43:431–473, May 1996.

[7] J. Gosler. Software protection: Myth or reality. In *Advances in Cryptology – CRYPTO 1985*.

[8] Jorge Guajardo, Sandeep Kumar, Geert-Jan Schrijen, and Pim Tuyls. Fpga intrinsic pufs and their use for ip protection. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 63–80. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74735-2_5.

[9] S. Kent. *Protecting Externally Supplied Software in Small Computers*. PhD thesis, Massachusetts Institute of Technology, 1980.

[10] Rishab Nithyanand and John H. Solis. A theoretical analysis: Physical unclonable functions and the software protection problem. Technical report, Sandia National Laboratories, Livermore, CA.

[11] R. S. Pappu. *Physical One Way Functions*. PhD thesis, Massachusetts Institute of Technology, 2001.

[12] U. Ruhrmair, J. Solter, and F. Sehnke. On the Foundations of Physical Unclonable Functions. Technical report, Cryptology ePrint Archive: Report 2009/277 http://eprint.iacr.org/2009/277.

[13] Stat Spotting. Software Piracy Statistics: 2011.

[14] C. Wang. *A Security Architecture for Survivability Mechanisms*. PhD thesis, University of Virginia, 2000.

DISTRIBUTION:

1	MS 9158	John H. Solis, 8961
1	MS 9158	Keith Vanderveen, 8961
1	MS 0899	Technical Library, 8944 (electronic copy)
1	MS 0161	Legal Intellectual Property, 11500
1	MS 0359	D. Chavez, LDRD Office, 1911

Sandia National Laboratories