SANDIA REPORT

SAND2011-6673
Unlimited Release
Printed September 2011

A Theoretical Analysis: Physical Unclonable Functions and The Software Protection Problem

Rishab Nithyanand, 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.

Sandia National Laboratories

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

SAND2011-6673
Unlimited Release
Printed September 2011

A Theoretical Analysis: Physical Unclonable Functions and The Software Protection Problem

Rishab Nithyanand
Stony Brook University
Stony Brook, NY, USA
rnithyanand@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. This property of unclonability is due to the accepted hardness of replicating the multitude of uncontrollable manufacturing characteristics and makes PUFs useful in 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 and show that traditional non-computational (black-box) PUFs cannot solve the problem against real world adversaries in offline settings.

Our contributions are the following: We provide two real world adversary models (weak and strong variants) and present definitions for security against the adversaries. We continue by proposing schemes secure against the weak adversary and show that no scheme is secure against a strong adversary without the use of trusted hardware. Finally, we present a protection scheme secure against strong adversaries based on trusted hardware.

Acknowledgments

We would like to thank Radu Sion for his feedback on an earlier version of this document.

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

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. This unclonability property comes from the accepted hardness of replicating the multitude of uncontrollable manufacturing characteristics.

In essence, 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 [10] and range from optical PUFs to silicon timing PUFs.

Although initially envisaged as a new tool for simple device identification and authentication, the attractiveness of PUF unclonability has greatly broadened the scope of possible applications. Current and emerging applications range from software protection and licensing to hardware tamper proofing and certified execution. However, the problem with the largest potential impact is *software protection* – software piracy costs the software industry a colossal loss of $51.4 billion annually [12].

Software protection in offline scenarios is extremely challenging because of *malicious hosts*. Malicious hosts have complete control, access, and visibility, over all software they execute, i.e., an extremely powerful adversary. This explains why most approaches fail to protect software from illegal tampering and/or duplication. Only well designed schemes, based on trusted devices (e.g., trusted hardware or servers), have had success in defending against malicious hosts. This prompts the question: Can PUF technology be used to build secure software protection schemes?

1.1 Contributions

In this paper, we investigate the effectiveness of PUFs for software protection from a theoretical standpoint. We make the case that traditional (black-box) PUFs cannot solve the software protection problem in offline settings. We define a traditional PUFs as devices that do not perform any computation, but behave solely as black box functions, i.e., given a challenge as input, output an unpredictable (but consistently repeatable) response.

Our contributions are the following: We provide two real world adversary models (weak and strong variants) and present definitions for security against each adversary. We continue by proposing schemes secure against the weak adversary and show that no scheme is secure against a strong adversary without the use of trusted hardware. Finally, we present a protection scheme secure against strong adversaries based on trusted hardware.

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 [13]. 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, unobfuscatable (by any code obfuscator). This implies, in its most extreme interpretation, that there does not exist a provably secure obfuscation algorithm that works on every program. Taking a new approach, 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 intellectual property 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 are several drawbacks to their approach: (1) it requires trusted hardware to remotely initialize the protection scheme and (2) it can only protect software with algebraic group functionality.

2 The Software Protection Problem

To properly tackle the software protection problem, it is important to accurately define the problem and any solution requirements. We re-iterate that rather than protecting software from trademark or copyright violations (i.e., software fingerprinting and watermarking) we aim to protect software from illegal execution and illegitimate duplication. To do this, we provide methods that ensure there is no feasible way to create illegal copies of the software that will run on a large number of machines.

In this section, we review theoretical preliminaries and provide answers to two fundamental questions: (1) What powers do real world adversaries have? (2) Given an adversary with certain powers, how can we produce software that is *uncrackable* without using trusted hardware or online trusted third parties?

2.1 Preliminaries

Formal PUF Definition

The formal definition of a PUF has been debated by several researchers recently [11]. For our purposes, we define a PUF as follows:

Definition 1. *A Physical Unclonable Function is a physical system with 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.

Programs as Turing Machines

A *Turing Machine* (TM) consists of a finite control, at least one infinite tape divided into cells, and a read and/or write head on each of the tapes. The finite control may be in one of many (but finite) number of states (Q). Each cell on the tape may contain one symbol from the alphabet of the machine (Σ), or a blank symbol (B). The tape head is capable of moving either left (L) or right

(R) from each cell. The TM begins in an *initial state* – q_0 and halts at a *halting state* – q_h. The state transition function (δ) determines how the machine changes state. The set of all inputs to the machine M that cause it to reach q_h is called its language $L(M)$. In our study of programs, we do not distinguish between (1) recursive and recursively enumerable languages or (2) deterministic and non-deterministic TMs. In general, a TM is assumed to be as powerful as a real machine, and can execute any program that a computing device can. We can therefore safely assume that there exists a TM for every computer program P.

Equivalence of Turing Machines

We say that two TMs M and M' are equivalent if $\forall x, M(x) = M'(x)$ – i.e., the functions computed by the two machines are identical on *every* input x. These two machines may have a different set of states and may also work with different state transition relations. Throughout this paper we use the terms *program* and *machine* interchangeably (also the notation M and P). We point out that the control flow graph of a program is an alternate visualization of the TM state transition diagram it represents.

Control Flow Graphs

A control flow graph (CFG) is a directed graph that denotes all execution paths traversed by a program during execution. The exact paths taken are usually dependent on user input and/or other branching conditions. Each node in the graph represents a linear block of code and each edge denotes the flow of control from one block to the next. Control flow graphs have two standard nodes: an entry node and an exit node. The entry node typically includes all instructions required for setting up the program execution environment, global declarations, etc. The exit block is where all execution halts – analogous to a TM halting state. When this block is reached we say that the program is complete.

3 Formalizing the Weak Adversary (W-ADV)

This model captures the scenario in which an adversary does not have access to the legitimate PUF. A real world analogue is a pirate that downloads or copies software to install and execute on its local systems.

3.1 PUF IP-Protection in the presence of a W-ADV

Consider the following experiment, defined for any PUF Intellectual Property-Protection (*IPP*) scheme, any adversary A, and any security parameter n (which determines the number of PUF challenges inserted in the output of *IPP*):

The W-ADV Experiment ($\mathbf{WADV}_{A,IPP}(n)$):

1. The *IPP* oracle picks at random two strings (that represent the Turing machines M and PUF) and produces the string M' which is of length $p(|M|)$, where $p(.)$ is some polynomial.
2. The adversary A is given as input n and the string of the machine M'.
3. The adversary A now outputs a string M''.
4. The output of the experiment is defined to be 1 if:
   $\forall x \in L(M)$, $[M''(x) = M'(x)]$ and $[\nexists C \in M''\ s.t\ f(R(C))$ is unknown$]$.
   The output is 0 otherwise. If $\mathrm{WADV}_{A,IPP}(n) = 1$, we say that A succeeded.

We say the adversary wins if M'' has the same functionality as M' and M'' has no PUF challenges in it whose responses (or function of responses) have not been guessed by the adversary. Note that this definition allows M'' to contain *zero* PUF challenges, i.e., have been removed by the adversary.

Let $E[\mathrm{WADV}_{A,IPP}(n) = 1]$ be the expected number of trials required by adversary A before he wins the the game (i.e., before $\mathrm{WADV}_{A,IPP}(n) = 1$).

Definition 2. *A PUF IP-Protection scheme IPP is said to be ε-secure in the presence of a W-ADV (or,* W-ADV ε-Secure*) if for all probabilistic polynomial time adversaries A there exists an ε exponential in the size of the program, such that:*

$$\boxed{E[(WADV_{A,IPP}(n)) = 1] \geq \varepsilon} \tag{1}$$

where n is the number of PUF challenges inserted in M' and the probability is taken over the random coins used by A, as well as the random coins used in the experiment (for choosing PUF challenges).

3.2 Formal Requirements of a W-ADV Secure IPP Scheme

Based on the definitions in Section 3.1, we now formally enumerate the requirements of a W-ADV secure IPP scheme:

- *(protected functionality)* The *protected* program P' must have the same functionality as the *original* program P. This requirement can be formalized as follows:

  If P is represented by the Turing machine M and P' is represented by the Turing machine M' then:

  $$\forall x \in L(M), M(x) = M'(x) \tag{2}$$

  Further, the program P' must be protected such that correct execution only occurs on a system with the attached PUF (i.e., for every challenge (C) in P', the response received ($R'(C)$) must be the expected response for that challenge ($R(C)$)). This requirement is essentially an if and only if pre-cursor to the above functionality requirement and can be formalized as follows:

  $$\boxed{\forall x \in L(M),\ \forall C \in M',\ M(x) = M'(x)\ \ \textit{iff}\ \ R(C) = R'(C)} \tag{3}$$

- *(non-trivial inversion)* There does not exist a polynomial time algorithm ADV such that, $ADV(M') \rightarrow M$. It should be hard for an adversary to create a functionally equivalent piece of software that does not perform PUF queries. This can be formalized as follows:

  $$\boxed{\forall ADV, Pr[ADV(M') = M''\ s.t\ M'' \equiv M\ and\ count(C \in M'') = 0] \leq \frac{1}{p(|M|)}} \tag{4}$$

3.3 A W-ADV Secure IPP Scheme Based on Control Flow Graphs

Let P and PUF be the inputs to the IPP scheme – i.e., P is the program to be protected and PUF is the PUF required for correct execution. Let G be the control flow graph of the program P where each node in the graph may represent a block of code with a bound on maximum size (b_{max}). This bound will be dependent on a security parameter n in the following way: $n \geq \lceil \frac{|P|}{b_{max}} \rceil$. Each of the n blocks is assigned an integer label: $\{b_1, ..., b_n\}$.

Construction 1. *The following construction causes the n node control flow graph G to be identified only when the PUF responses to challenges are correct. Given an incorrect PUF, the control flow graph resembles a complete graph with n nodes.*

1. *At the exit point of every block b_i, a challenge is inserted by the vendor as follows:*

   (a) *If the original control flow graph G of the program P contains an edge from node b_i to b_j, then a challenge C_i is picked, such that, $f(R(C_i))$ equals the integer label of b_j.*

(b) *The challenge is inserted as: (1) an unconditional branching statement,* e.g., goto $f(R(C_i))$*, or (2) part of an existing conditional branching statement,* e.g., if $(a == b)$ then goto $f(R(C_i))$.

2. *The above procedure is repeated for every edge in the original control flow graph G.*

Properties and Security

The resulting program P' has the following properties:

- *(non-trivial inversion: complete CFG in the presence of a W-ADV)* The CFG of P' appears to be a complete graph because, at the exit point of a given block, the adversary is unaware of the correct $f(R(C_i))$ value and cannot do significantly better than guessing. This means that each of the remaining $n-1$ blocks is equally valid as the next node in the CFG. Therefore, the CFG has edges between all possible blocks and forces the adversary to guess the next block – a correct guess occurs with probability $\frac{1}{n-1}$ for each block.
- *(protected functionality: correct CFG in the presence of PUF)* In the presence of PUF, the program P' and its control flow graph G' have the same functionality and structure as the program P and its graph G, respectively. This is because the correct response $f(R(C_i))$ is given for every challenge C_i in block b_i with probability equal to 1.

An example of the difference in the control flow graph of P' with and without the correct PUF is illustrated in Figure 1.

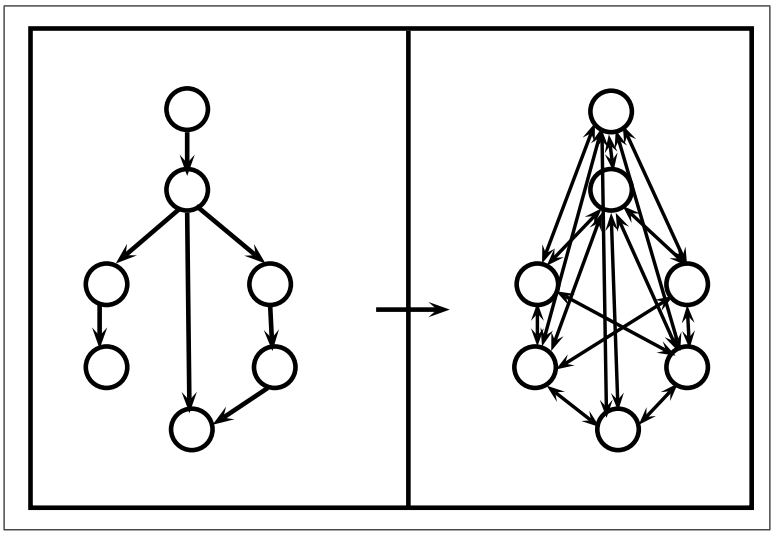

Figure 1. Example $CFG(P')$ on $PUF \rightarrow CFG(P')$ on PUF'

Theorem 1. *Construction 1 is W-ADV Secure.*

Proof. To show that Construction 1 is W-ADV secure according to Definition 2, we must show that the expected number of trials before an adversary may win the W-ADV experiment is at least exponential in n. We know that the probability of an adversary correctly guessing the next block to enter from block b_i is $\frac{1}{n-1}$ (if we assume no loops, allowing loops only makes our case stronger). The probability of correctly guessing a single path with length m is $\frac{1}{(n-1)^m}$. However, since the number of paths (and path lengths) in P' are unknown to the adversary, an exhaustive search over all n blocks is required to discover all possible paths. The probability of n consecutive correct guesses for the values of $f(R(C_i))$ is $\frac{1}{(n-1)^n}$. Therefore, the expected number of trials before the adversary is able to guess all $f(R(C_i))s$ correctly is $(n-1)^n$.

□

Limitations

While our scheme is theoretically secure, it does suffer from what we like calling a *reality shock*. In the real world, it is likely that a program will crash if control is randomly transferred from one block to another. If this behavior occurs with high probability (≈ 1), an adversary can reconstruct the correct control flow in $O(n^2)$ trials. We argue, however, that methods exist to prevent crashing on unexpected control flow, such as, global declarations in the entry block, choosing very fine grained block granularity, etc.

Furthermore, the deterministic PUF responses in the above construction make it is possible to create a crack of P' given access to the real PUF using the attack described in Section 4.2. In the following section, we show how to convert the above scheme into one that is secure against an strong adversary using cryptographic primitives and a trusted computing board. We also illustrate reasons for why we believe it is impossible to achieve security against a strong adversary without a trusted computing board.

4 Formalizing the Strong Adversary (S-ADV)

This model captures the scenario in which an adversary has limited time access to the legitimate PUF. In the real world, this captures effectively the scenario where an institution buys a single software license, studies the software, and attempts to create a cracked version to distribute to multiple systems.

4.1 PUF IP-Protection in the presence of a S-ADV

The basic idea behind a S-ADV is that the adversary A is allowed to adaptively query the PUF used by the IPP algorithm. This is formalized by allowing A to interact freely with the PUF oracle as a black-box that returns responses (or functions of responses) to challenges sent to it by A. The following experiment is defined for any PUF IP-Protection scheme IPP, any adversary A, and any security parameter n (the number of PUF challenges inserted in the output of IPP).

The S-ADV Experiment ($\mathbf{SADV}_{A,IPP}(n)$):

1. The IPP oracle picks at random two strings (that represent the Turing machines M and PUF) and produces the string M' which is of length $p(|M|)$, where $p(.)$ is some polynomial.
2. The adversary A is given as input n, access to $O^{f(PUF)}$, and the string of the machine M'.
3. The adversary A continues to have oracle access to the machine PUF. It then outputs a string M''. Let C' be the set of all queries sent to the PUF oracle $O^{f(PUF)}$.
4. The output of the experiment is defined to be 1 if:

   $\forall x \in L(M)$, $[M''(x) = M'(x)]$ and $[\nexists C \in M''\ s.t\ f(R(C))$ is unknown$]$.

   The output is 0 otherwise. If $\mathrm{SADV}_{A,IPP}(n) = 1$, we say that A succeeded.

We say that the adversary has won if M'' has the same functionality as M' and if M'' has no PUF challenges in it whose responses (or function of responses) have not been guessed or learned from the PUF oracle. Note that this allows M'' to contain *zero* PUF challenges, i.e., have been removed by the adversary.

Let $E[\mathrm{SADV}_{A,IPP}(n) = 1]$ be the expected number of trials required by adversary A before he wins the the game (i.e., before $\mathrm{SADV}_{A,IPP}(n) = 1$).

Definition 3. *A PUF IP-Protection scheme IPP is said to be ε-secure in the presence of a S-ADV (or,* S-ADV ε-Secure*) if for all probabilistic polynomial time adversaries A there exists an ε exponential in the size of the program, such that:*

$$\boxed{E[(SADV_{A,IPP}(n)) = 1] \geq \varepsilon} \tag{5}$$

where n is the number of PUF challenges inserted in M' and the probability is taken over the random coins used by A, as well as the random coins used in the experiment (for choosing PUF challenges).

Clearly, if an *IPP* scheme is secure against a S-ADV, it is also secure against a W-ADV. This holds because the WADV experiment is a special case of the SADV experiment in which the adversary A does not access the PUF oracle at all.

4.2 Discussion

At first sight, it appears that security against a S-ADV is impossible to achieve. In particular, consider an adversary that gets as input a program P'. Since the adversary has oracle access to $O^{f(PUF)}$, it can request responses (or a function of the responses) for all challenges in P'. This makes it possible to create P'' by simply replaying the recorded responses. Such an attack easily breaks the protection provided by IPP, since Equation 5 is now:

$$\boxed{E[(SADV_{A,IPP}(n)) = 1] = 1} \tag{6}$$

We conclude that no *IPP* scheme can be secure against an S-ADV if the PUF challenges in P' are deterministic. Avoiding this issue requires: (1) PUF responses are no longer dependent solely on the challenge, or (2) non-deterministic PUF challenges. However, the first method violates our fundamental assumption that a PUF is a non-computational (black-box) device.

4.3 Strong Adversarial Approaches

In general, an S-ADV A may take one of the following two approaches in order to create a cracked version P'' of a protected program P'.

1. A may execute the program for a given input on the legitimate PUF and then observe the log of executed blocks. He then removes all instances of the PUF challenges. This essentially creates the *cracked* version P'' for the single execution path that was reached with the supplied input. Here, the difficulty of creating the crack is expressed in terms of the number of paths present in the state transition diagram of the machine P'.

2. A may instead scan the program P' and store every challenge embedded in it. The responses to each of these challenges are then stored in a table of (C,R) pairs – thereby, virtualizing the $useful$ part of the PUF. Here, the difficulty of creating the crack is expressed in terms of the probability of the adversary guessing correctly all responses to challenges presented by the software.

Based on the above two approaches, the lower bound on the number of iterations required by the adversary to create a cracked version of the software is: *min(# paths, # iterations required for guessing or learning all correct PUF responses)*. We point out that the number of paths in the control flow graph of the protected program P' (or, in the state transition diagram for M') is controlled by the software developer. Various programming techniques, such as, obfuscation, can increase CFG branching factor. These techniques are beyond the scope of our paper. Instead, we focus on maximizing the number of iterations required before an adversary can guess all responses to the challenges in P' correctly.

4.4 An Impossibility Conjecture

We now show that it is impossible to build a S-ADV secure IPP scheme without using trusted hardware for secure storage and/or processing. We first present an informal argument to shows intuitively why we believe this to be true. Note that the following arguments also apply to software obfuscation, whitebox cryptography, and software watermarking.

Conjecture 1. *There cannot exist a S-ADV secure IPP scheme in offline settings without trusted hardware.*

Reasoning. It is clear from the above requirements that there needs to be some type of randomness involved in the selection of challenges. We now 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 that (1) multiple state transitions may exist for entries in the state transition function, and (2) transitions are made based on probabilities determined by a random tape R which consists 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 be read-only (i.e., it is not write enabled). However, it is impossible to enforce this requirement in the purely offline setting without trusted hardware – every tape is write-enabled. Since the random tape is write enabled, an adversary may rewrite the tape with the bits y to enforce a certain set of challenges on every iteration of PTM. The end result is a deterministic Turing machine $PTM(x)$ rather than the desired probabilistic machine $PTM(x,y)$. This enables the adversary to launch the attack described in Section 4.2 and win the S-ADV experiment after just a single trial.

4.5 A S-ADV Secure Scheme Using Trusted Hardware

Let P and PUF be the inputs to the IPP scheme, i.e., P is the program to be protected and PUF is the PUF required for correct execution. Let G be the control flow graph of the program P where each node in the graph may represent a block of code with a bound on maximum size (b_{max}). This bound will be dependent on a security parameter n in the following way: $n \geq \lceil \frac{|P|}{b_{max}} \rceil$. Each of the n blocks is assigned an integer label: $\{b_1, ..., b_n\} \in \{1, ..., n\}$. We make the following assumptions about the trusted hardware: (1) It is capable of storing $O(n \log n)$ bits in secure memory (for the entire lifetime of the program), and (2) It is capable of performing strong pseudo-random permutation operations (*s-PRP*).

Strong Pseudo Random Permutations

A function F:$\{0,1\}^l \times \{0,1\}^l \rightarrow \{0,1\}^l$ is a keyed s-PRP if:

- For every k, $F_k(\cdot)$ is a one-to-one function.
- Given k, x there exist efficient functions for computing $F_k(x)$ and its inverse $F_k^{-1}(x)$.
- An adversary with access to the inverse function oracle cannot distinguish between $F_k(\cdot)$ and a randomly chosen permutation.

We build our *IPP* scheme based on the assumption that strong pseudo random permutations exist – a conjecture widely believed to be true. In our construction, the key k for the s-PRP $F_k(\cdot)$ is stored in secure memory.

IPP-Program State

The IPP-Program State is initialized to $\{b_1, ..., b_n\}$. After the j^{th} execution, the state is updated to $\{F_k^{(j)}(b_1), ..., F_k^{(j)}(b_n)\}$ where F_k is a keyed s-PRP function. We say that the value $F_k^{(j)}(b_i)$ is the label assigned to the block b_i in the j^{th} iteration.

IPP-PRP Tables

In the secure memory provided by the trusted hardware, a 3-tuple-n-record table (called the IPP-PRP Table) is stored. There is a record for every block in P containing the following fields:

- *Block Index:* The block index of a block i is the initial label b_i assigned to it. This tuple is the primary key to the PRP-IPP table and does not change during the entire lifetime of the program P'.

- *PRP Index:* The PRP index of a block i is the label assigned to it by the IPP-Program state described above. The value is unique for each block, however it changes on each iteration in accordance with the IPP-Program state.

- *Challenge Set:* The challenge set for a block i is a large but finite set of challenges that have the following property: for every challenge in the challenge set, $f(R(C)) = F_k^{(j)}(b_i)$. Notice that the input and output domains of the function $F_k(\cdot)$ are the same, therefore, only n challenge sets need to be collected by the vendor of P'. The set, as with the PRP index also changes with every update of the IPP-Program state.

Construction 2. *Our construction causes the n node control flow graph G to be identified only when the PUF responses to challenges are correct. Given an incorrect PUF, the control flow graph resembles an n node complete graph. Note that this construction illustrates only one instance of the program. The blocks of the program are relabeled on each iteration (or instance), as required by the software vendor. The same construction applies after relabeling.*

1. *At the exit point of every block with* block index *– b_i, challenges are inserted by the vendor as follows:*

   (a) *If the control flow graph G of the program P contains an edge from a block with index b_i to a block with index b_j, then a challenge C is selected at random from the* challenge set *in the record that contains b_j as the* block index. *Providing the expected response to this challenge transfers control to the correct block in the re-labeled control flow graph.*

   (b) *The challenge is inserted as: (1) an unconditional branching statement,* e.g., goto $f(R(C))$*, or (2) part of an existing conditional branching statement,* e.g., if $(a == b)$ then goto $f(R(C))$ else goto $f(R(C'))$.

2. *The above procedure is repeated for every edge in the control flow graph G.*

Properties and Security

- *(non-trivial inversion: complete CFG in the presence of a S-ADV)*

  The control flow graph of the program P' appears as a complete graph because: On a new instance of the program (due to relabeling of nodes in the control flow graph), at the exit point of a given block, the adversary is unaware of the correct (re-labeled) value of $f(R(C_i))$ and cannot do significantly better than guessing its value. Correctly guessing the next block occurs with probability $\frac{1}{n-1}$.

- *(protected functionality: correct CFG in the presence of PUF)* In the presence of PUF, the program P' and its control flow graph G' have the same functionality and structure as the program P and its graph G, respectively. This is because the correct response $f(R(C_i))$ is given for every challenge C_i in block b_i with probability equal to 1.

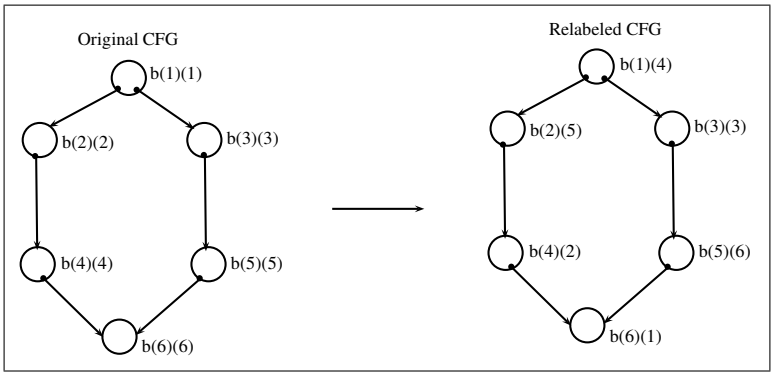

Figure 2. (example) Initial CFG on iteration $0 \rightarrow$ Relabeled CFG on iteration 1. The value in the first parenthesis is the block index, the second is the PRP index. The exit point of each block contains a challenge C such that $f(R(C)) == PRP\ index(b_j)$.

Theorem 2. *Construction 2 is S-ADV Secure*

Proof. To show that Construction 2 is S-ADV secure, according to Definition 3, we must show that the expected number of trials before an adversary may win the S-ADV experiment is at least exponential in n. If $F_K(\cdot)$ is a strong pseudo random permutation (as defined in subsection 4.5), on a new program instance, we know that the probability of an adversary correctly guessing the label of the next block to enter from block b_i is $\frac{1}{n-1}$ (if we assume no loops, allowing loops only makes our case stronger). There are n blocks, therefore the probability of n consecutive correct guesses for the values of $f(R(C_i))$ is $\frac{1}{(n-1)^n}$. Therefore, the expected number of trials before the adversary is able to guess all $f(R(C_i))s$ correctly is $(n-1)^n$. □

Discussion: Challenge Set Size vs. Security

The size of each challenge set directly corresponds to the number of iterations an adversary must run (on a particular path of the control flow graph) before being able to use the (protected) software on a *virtualized* PUF.

Therefore, the challenge set size must be large enough to prevent "brute-force" attacks by the adversary. This can also be enforced by ensuring that licenses are tied to specific number of uses rather than unlimited use. After each use (i.e., instantiation) of the program, the challenge entry of the challenge set is deleted. Eventually, the number of entries for some challenge set will reach *null*, causing the program to terminate abruptly. At this point, the user will be required to request more challenge sets from the software vendor. The size of the challenge set and per-usage licenses are an important security parameter/policy left in the hands of software vendors.

5 Rethinking the Software Protection Problem

As is clear from Section 4.4, it is impossible to achieve security against an S-ADV without a trusted entity (e.g., trusted hardware or online server). This is not reasonable since security against the S-ADV can be achieved without using a PUF while using a trusted party (e.g., see oblivious RAM). We reiterate that our goal is to find a feasible offline solution that does not require additional trusted hardware. To this end, we recognize the need to 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 to the software protection problem.

5.1 Failure of Traditional PUFs

The main reason for the failure of traditional PUFs is the impossibility of supplying random challenges to the PUF from a deterministic program. Further, the PUF is only a peripheral device connected to the device executing the program via some bus, and in the hostile environment (modeled by the S-ADV), any information flow through the bus is known and monitored by the adversary. This allows an adversary to easily replicate/virtualize the PUF and makes them unusable against the S-ADV.

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 in the execution of the software 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.

5.2 Failure of Traditional Computing Models

Unfortunately, traditional Turing machines or RAM computing models are not useful with the software protection problem because intrinsic features and randomness (such as timing delays and bit errors) cannot be sufficiently modeled. Any future attempts to find a purely PUF based solution to the offline protection problem should rely on a systems oriented toolkit (recall that theoretical solutions cannot exist).

6 Conclusions and Future Work

PUFs have been envisioned as being applicable to practical problems, such as, hardware authentication, certified execution, and most notably software protection. However, current approaches attempting to use PUFs for offline hardware authentication and software protection are vulnerable to virtualization attacks

However, we believe that using IP-PUFs as can reduce such attacks significantly by continuously authenticating the device implicitly and transparently. Further, this method of authentication is useful for software protection by intertwining software and a computing device (e.g., by inserting race conditions that resolve correctly only on the correct device). This approach makes it increasingly difficult for an adversary unhooking functionality from the PUF. The development of such a PUF is the logical culmination for this project and will be part of our future work.

In conclusion, we first showed that traditional non-computational (black-box) PUFs are not useful in solving the software protection problem in offline scenarios. We provided two real world adversary models and proposed schemes secure both (using trusted hardware in the strong case). Our results show that incorporating PUFs as a method for software protection will require systems based approaches and research methods.

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

[11] 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.

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

[13] 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 0359	D. Chavez, LDRD Office, 1911

Sandia National Laboratories