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 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 are easy to measure but hard to clone. The unclonability property 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 and licensing, and certified execution. In this paper, we investigate the effectiveness of PUFs for software protection in hostile offline settings.**

We show that traditional non-computational (black-box) PUFs cannot solve the software protection problem in this context. We provide two real-world adversary models (weak and strong variants) and security definitions for each. We propose 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.

***Keywords*-physical unclonable functions, PUFs, physical one-way functions, software protection, intellectual property protection**

I. Introduction

Physical Unclonable Functions (PUFs) or Physical One Way Functions (P-OWFs) are physical systems whose responses to input stimuli are easy to measure, within reasonable error bounds, but hard to clone. In essence, PUFs hide their *secrets* in circuit characteristics rather than in digitized form. On different input stimuli (challenges) a PUF circuit exposes certain measurable and persistent characteristics (responses). The unclonability property comes from the accepted hardness of replicating the multitude of uncontrollable manufacturing characteristics. Several varieties of PUFs have been proposed since being introduced by Pappu in [1] and range from optical PUFs [1], [2] to silicon timing PUFs [3], [4].

Although initially envisaged as a new device identification and authentication tool, the attractiveness of PUF unclonability has greatly broadened the scope of possible applications. Current and emerging applications include software protection and licensing [5]–[7], hardware tamper proofing [8], [9], and certified execution [10], [11]. However, the problem with the largest commercial impact is *software protection*. Software piracy costs the software industry billions of dollars annually in lost revenue.

Protecting software in offline scenarios is extremely challenging because of *malicious hosts*. Malicious hosts have complete control, access, and visibility, over all software they execute. This makes for an extremely powerful adversary and explains why most approaches fail to protect software from illegal tampering and duplication. The main security issue with using PUFs in this hostile context is dealing with PUF replay and virtualization attacks [6] – also referred to as OORE (Observe Once, Run Everywhere) attacks. Only well designed schemes based on trusted devices (e.g., trusted hardware or servers) can have any success. This raises interesting research questions: Is it possible to use PUF technologies to build secure software protection schemes? What would such schemes look like?

A. Contributions

In this paper, we seek to answer the above questions by investigating the effectiveness of PUFs for software protection from a theoretical standpoint. While the formalization may seem verbose, it allows us to study the fundamental behavior of PUFs and to draw conclusions independent of specific implementations. Furthermore, the framework we present, as we argue later, is not far from reality.

Our main contribution is showing that traditional (black-box) PUFs *cannot* solve the software protection problem in offline settings. Traditional PUFs are defined 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.

We also contribute 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 propose a protection scheme secure against strong adversaries based on trusted hardware.

B. Related Work

The first work geared towards the anti-piracy and software protection problem was in 1980 by Kent [12]. Kent suggested the use of tamper resistant trusted hardware and encrypted programs and was the first to differentiate the trusted host problem from the trusted code problem. Gosler [13] 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 OORE attacks.

Cohen [14] 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 techniques were later proposed by Collberg *et al* [15] and Wang [16]. Finally, Goldreich and Ostrovsky provided the first theoretical analysis and foundation to the software protection problem [17]. The basic approach hides/obfuscates data access patterns in conjunction with trusted hardware to prevent illegal software replication.

More recently, Boaz Barak *et al* [18] completed a theoretical analysis of software obfuscation techniques. Their contribution was an interesting negative result that implied, in its most extreme interpretation, that there does not exist a provably secure obfuscation algorithm that works on all possible programs. Taking a new approach, Chang and Atallah [19] proposed a scheme that prevented software tampering using a set of inter-connected (code) guards programmed to perform code verification and repairs.

The advent of PUFs has led to several proposals for their use in software protection. Most notably, Guajardo *et al.* [5], proposed an FPGA based intellectual property protection scheme that relied on SRAM PUFs. However, SRAM PUFs are not ideal due to the possibility of an exhaustive read out attack. Atallah *et al.* [6] proposed inter-twining software functionality directly with the PUF. However, their approach requires trusted hardware for remote initialization and only protects software with algebraic group functionality.

C. Organization

The remainder of this paper is organized as follows: Section II explains the software protection problem and reviews the concepts needed to understand this paper: PUFs, Turing Machines (TMs), and Control Flow Graphs (CFGs). In Section III we introduce a weak adversary (W-ADV) model along with a software protection scheme secure against this adversary. Section IV extends the previous model to capture a strong adversary (S-ADV) and argues why trusted hardware is required to protect against such adversaries. We then present a software protection scheme based on trusted hardware secure against a S-ADV. Section V, presents future directions for research and motivates the need to rethink current approaches to software protection. Finally, we summarize our conclusions in Section VI.

II. 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 duplication.

In our model, the adversary's goal is to create a duplicate program with *identical* functionality. At first glance this seems unrealistic – in most practical situations it is sufficient for a duplicated program to have similar or partial behavior. It is common to see pirated software that only supports a subset of features or that only occasionally crashes.

However, if we can devise a generic method capable of protecting small code block sizes, then this method can be extended to larger programs with several protected "features". In the ideal situation, it should be difficult for an adversary to identify a specific feature and remove it without affecting other features, i.e., duplication of all but one feature is also difficult.

We now review theoretical preliminaries and answer the questions: (1) What powers do real-world adversaries have? (2) How do we protect software against real-world adversaries without using trusted hardware or online trusted third parties?

A. Preliminaries

Formal PUF Definition: Although the formal definition of a PUF has been under debate recently [20]–[23], 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 (PUF'), it is infeasible for an adversary to build another system (PUF'') – 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 – i.e., for a hardware device to be unclonable, it must possess randomness. We acknowledge that the above definition does not capture the notion of noise in PUF responses (as in [23]). The effect of noise on persistence can be captured by introducing a threshold parameter (α) and require that the response to some challenge occurs with a probability of at least α (after error correction). In this paper, we do not address noisy PUF responses, but assume interaction with a PUF with a threshold parameter $\alpha = 1$.

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. This allows us to 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 (denoted $M \equiv M'$) 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.

Unfortunately, it is not always possible to confirm complete equivalence since the language of a machine $|L(M)|$ may be infinite. For this reason, we cannot require strict equivalence and assume that the input set size is finite and covers all features of the program.

Throughout this paper we use the terms *program* and *machine* (and notations M and P) interchangeably.

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 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.

Note that the control flow graph of a program is an alternative visualization of the TM state transition diagram it represents.

III. 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 an adversary who makes an

Table I
SUMMARY OF NOTATION IN SECTION III

Notation	Definition
IPP	Intellectual Property Protection scheme
A (or) ADV	A polynomially (time) bounded adversary
n	Security parameter determining the number of PUF challenges inserted in a program
$p(\cdot)$	A positive polynomial function
$WADV_{A,IPP}(n)$	The output of the WADV game with A, IPP, n as inputs.
$E[T_{WADV_{A,IPP}}(n)=1]$	Expected number of trials of the WADV game before $WADV_{A,IPP}(n) = 1$.
N	Number of blocks in a program
b_i	Integer label of block i
$f(R(C))$	Function f applied to $R(C)$
$CFG(P)$	The control flow graph representing program P

exact duplicate of software to install and execute on its local systems.

A. 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 (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' embedded with n PUF challenges which is of length $p(|M|)$, for some polynomial $p(\cdot)$. We denote an embedded challenge C as $C \in M'$.
2) The adversary A is given as input n and the string of the machine M'.
3) The adversary A outputs a string M''.
4) The experiment output is defined to be 1 if:
   $\forall x \in L(M)$, $[M''(x) = M'(x)]$ *and*
   $[\nexists C \in M''$ *s.t* $R(C)$ is unknown$]$

   Otherwise the output is 0. We say that A succeeded if WADV$_{A,IPP}(n)$ = 1.

The adversary wins the experiment 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 for M'' to contain *zero* PUF challenges, i.e., have been removed by the adversary.

It may seem odd to model "unclonable" PUFs as random strings that are inherently clonable. However, we justify this decision by arguing that a deterministic challenge/response PUF can be viewed as an exponentially large lookup table. An adversary with a polynomial amount of storage space cannot duplicate the PUF string in its entirety. i.e., it is impossible for an adversary to perform a read-out attack and virtualize the entire domain space of the PUF.

Let $E[T_{WADV_{A,IPP}}(n) = 1]$ be the expected number of trials required by adversary A before winning the game, where a trial is a single execution of M' for some x of the adversary's choosing.

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:*

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

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).

B. Formal Requirements of a W-ADV Secure IPP Scheme

Based on the definitions in Section III-A, 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:

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

Further, the program P' must be protected such that correct execution only occurs on a system with the attached PUF, i.e., for every embedded challenge $C \in$

P', the response $R'(C)$ must be the expected response for that challenge. This requirement is essentially an if and only if pre-cursor to the above functionality requirement:

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

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

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

C. W-ADV Secure IPP Scheme Based on Control Flow Graphs

Let P and PUF be the inputs to the IPP scheme, where P is the program to be protected and PUF the PUF oracle required for correct execution. Let G be the control flow graph of program P where each node represents a block of code. The size of each code block is dependent on the security parameter n in the following way: Let $N \geq \lceil \frac{|P|}{n} \rceil$ and assign each code block 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 pick challenge C_i such that $f(R(C_i))$ equals the integer label of b_j.*
    b) *The challenge is inserted as an unconditional branching statement,* e.g., goto $f(R(C_i))$*, or as 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. Thus, each of the remaining $N-1$ blocks is equally valid as the next node in the CFG. This creates 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 the PUF oracle, 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 oracle is illustrated in Figure 1.

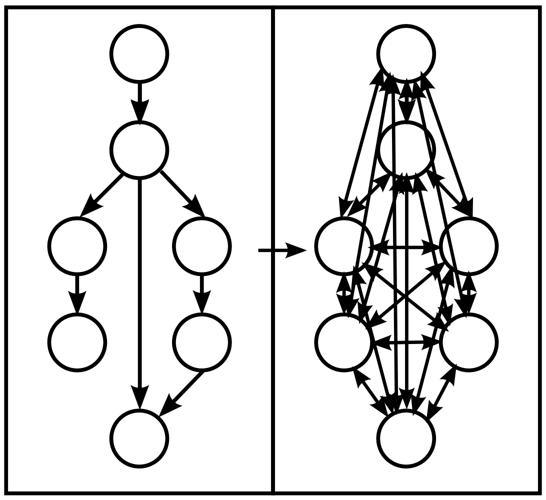

Figure 1. Example $CFG(P')$ on $PUF \rightarrow CFG(P')$ on PUF'

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 of 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 $f(R(C_i))$ correctly is $(N-1)^N$.

■

Limitations: While our scheme is theoretically secure, it does suffer from what we call 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. However, methods exist to prevent crashing on unexpected control flow, such as, global variable declarations, choosing very fine grained block granularity, etc.

Furthermore, the deterministic PUF responses in the above construction make it possible to create a P' given access to the real PUF oracle using the attack described in Section IV-B. In the following section, we show how to convert the above scheme into one that is secure against a 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.

IV. Formalizing the Strong Adversary (S-ADV)

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

Table II
Summary of notation in Section IV

Notation	Definition
$p(\cdot), p'(\cdot)$	Positive polynomial functions
$SADV_{A,IPP}(n)$	The output of the SADV game with A, IPP, n as inputs.
$E[T_{SADV_{A,IPP}}(n) = 1]$	Expected number of trials of the SADV game before $SADV_{A,IPP}(n) = 1$.
PTM_y	A probabilistic Turing machine with random tape set to string y.
$PTM_y(x)$	A PTM with input x and the random tape set to y.
$L(PTM_y)$	The language of PTM_y.
$F_k(\cdot)$	A strong pseudo-random permutation with key k.
$F_k^j(x)$	$F_k(\cdot)$ applied j times on input x.

A. PUF IP-Protection in the presence of a S-ADV

The basic idea behind a S-ADV is that the adversary is allowed to adaptively query the PUF oracle 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 issued by A. The following experiment is defined for any PUF IP-Protection scheme IPP, any adversary A, and any security parameter n (number of PUF challenges inserted in the output of IPP).

The S-ADV Experiment (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' embedded with n PUF challenges which is of length $p(|M|)$, for some polynomial $p(\cdot)$. We denote an embedded challenge C as $C \in M'$.
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 experiment output 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$]$

   Otherwise, the output is 0. We say that A succeeded if SADV$_{A,IPP}(n)$ = 1.

The adversary wins the experiment 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 for M'' to contain *zero* PUF challenges, i.e., have been removed by the adversary.

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

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:*

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

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.

B. 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:

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

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 that are not dependent solely on the challenge, or (2) non-deterministic PUF challenges. However, the first method violates our assumption that a PUF is a non-computational (black-box) device.

C. 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 observe the log of executed blocks. He can 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 the minimum between the number paths in P and the number of 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 but are beyond the scope of our paper. Instead, we focus on maximizing the number of iterations required before an adversary correctly guesses all responses to the challenges in P'.

D. An Impossibility Conjecture

We now argue why 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 show intuitively why we believe this to be true. Our arguments are also applicable 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 which 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 $PTM_y(x)$ 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 $TM(x)$ rather than the desired probabilistic machine $PTM_y(x)$. This enables the adversary to launch the attack described in Section IV-B and win the S-ADV experiment after just a single trial.

E. A S-ADV Secure Scheme Using Trusted Hardware

Let P and PUF be the inputs to the IPP scheme, where 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 represents a block of code. As before, code block size is bounded by the security parameter n in the following way: Let $N \geq \lceil \frac{|P|}{n} \rceil$ with each block assigned an integer label: $\{b_1, ..., b_N\} \in \{1, ..., N\}$. We assume the trusted hardware can store $O(N \log N)$ bits in secure memory (for the entire lifetime of the program) and can perform 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 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.*

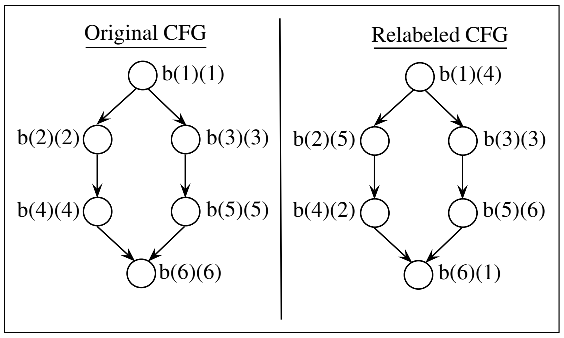

Figure 2. (example) Initial CFG on iteration 0 $\rightarrow$ Relabeled CFG on iteration 1. The value in the first parenthesis is the block index, the second is the PRP index. The exit point of each block contains a challenge C such that $f(R(C)) == PRP\ index(b_j)$.

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.

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, 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.

V. Rethinking the Software Protection Problem

Section IV-D argues why it is impossible to achieve security against a S-ADV without a trusted entity (e.g., trusted hardware or online server). Unfortunately, this does not meet our original goal of finding a feasible offline solution without additional trusted hardware. This requires that 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 a solution to the software protection problem.

A. 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.

B. 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.

VI. 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

We believe that using IP-PUFs 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 to unhook software 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 methodologies.

Acknowledgments

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 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.

References

[1] R. S. Pappu, "Physical one way functions," Ph.D. dissertation, Massachusetts Institute of Technology, 2001.

[2] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld, "Physical one-way functions," *Science*, vol. 297, no. 5589, pp. 2026–2030, 2002.

[3] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, "Silicon physical random functions," in *Proceedings of the 9th ACM conference on Computer and communications security*, ser. CCS '02. New York, NY, USA: ACM, 2002, pp. 148–160.

[4] D. Lim, J. Lee, B. Gassend, G. Suh, M. van Dijk, and S. Devadas, "Extracting secret keys from integrated circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 10, pp. 1200 –1205, oct. 2005.

[5] J. Guajardo, S. Kumar, G.-J. Schrijen, and P. Tuyls, "Fpga intrinsic pufs and their use for ip protection," in *Cryptographic Hardware and Embedded Systems - CHES 2007*. Springer Berlin / Heidelberg, 2007, pp. 63–80.

[6] M. J. Atallah, E. D. Bryant, J. T. Korb, and J. 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*. New York, NY, USA: ACM, 2008, pp. 45–48.

[7] J. Guajardo, S. Kumar, G.-J. Schrijen, and P. Tuyls, "Brand and ip protection with physical unclonable functions," in *IEEE International Symposium on Circuits and Systems, ISCAS 2008.*, may 2008, pp. 3186 –3189.

[8] E. Ozturk, G. Hammouri, and B. Sunar, "Physical unclonable function with tristate buffers," in *IEEE International Symposium on Circuits and Systems, ISCAS 2008.*, may 2008, pp. 3194 –3197.

[9] P. Tuyls, G.-J. Schrijen, B. Škorić, J. van Geloven, N. Verhaegh, and R. Wolters, "Read-proof hardware from protective coatings," 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. 369–383.

[10] P. Tuyls and B. Škorić, "Strong authentication with physical unclonable functions," in *Security, Privacy, and Trust in Modern Data Management*, 2007, pp. 133–148.

[11] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, "Controlled physical random functions," in *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, 2002, pp. 149 – 160.

[12] S. Kent, "Protecting externally supplied software in small computers," Ph.D. dissertation, Massachusetts Institute of Technology, 1980.

[13] J. Gosler, "Software protection: Myth or reality," in *Advances in Cryptology – CRYPTO 1985.*

[14] F. B. Cohen, "Operating system protection through program evolution," *Comput. Secur.*, vol. 12, pp. 565–584, October 1993.

[15] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Tech. Rep. 148, Jul. 1997.

[16] C. Wang, "A security architecture for survivability mechanisms," Ph.D. dissertation, University of Virginia, 2000.

[17] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, pp. 431–473, May 1996.

[18] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *Advances in Cryptology — CRYPTO 2001*, ser. Lecture Notes in Computer Science, J. Kilian, Ed. Springer Berlin / Heidelberg, 2001, vol. 2139, pp. 1–18.

[19] H. Chang and M. J. Atallah, "Protecting software code by guards," in *Digital Rights Management Workshop*, 2001, pp. 160–175.

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

[21] R. Maes and I. Verbauwhede, "Physically unclonable functions: A study on the state of the art and future research directions," in *Towards Hardware-Intrinsic Security*, ser. Information Security and Cryptography, A.-R. Sadeghi and D. Naccache, Eds. Springer Berlin Heidelberg, 2010, pp. 3–37.

[22] P. Tuyls, B. Skoric, S. Stallinga, T. Akkermans, and W. Ophey, "An information theoretic model for physical uncloneable functions," in *International Symposium on Information Theory, 2004.*, june-2 july 2004.

[23] F. Armknecht, R. Maes, A. Sadeghi, F.-X. Standaert, and C. Wachsmann, "A formalization of the security features of physical functions," in *2011 IEEE Symposium on Security and Privacy (SP).*, may 2011, pp. 397 –412.