POSTER: Making the Case for Intrinsic Personal Physical Unclonable Functions (IP-PUFs)

[Extended Abstract]

Rishab Nithyanand
Stony Brook University
rnithyanand@cs.stonybrook.edu

Radu Sion
Stony Brook University
sion@cs.stonybrook.edu

John Solis
Sandia National Laboratories
jhsolis@sandia.gov

ABSTRACT

Physical Unclonable Functions (PUFs) are physical systems whose responses to input stimuli (i.e., challenges) are easy to measure but difficult 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 authentication, software protection/licensing, and certified execution.

In this abstract, we claim that any multi-core computer is usable as a timing-PUF and can be measured via simple benchmarking tools (i.e., no specialized hardware required). We investigate several characterstics of standard off-the-shelf computers and present initial experimental results justifying our claim. Additionally, we argue that PUFs which are intrinsically involved in computations over sensitive data are preferable to peripheral device PUFs – especially for intellectual property protection and continuous device authentication.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Authentication*; K.5.1 [**Legal Aspects of Computing**]: Hardware and Software Protection—*Licensing*

General Terms

Experimentation, Security

Keywords

Physical Unclonable Functions, Authentication, Software Protection, Hardware

1. INTRODUCTION

Physical Unclonable Functions rely on hiding *secrets* in circuit characteristics rather than in digitized form. On different input stimuli (i.e., challenges) a PUF circuit exposes certain measurable and unpredictable (yet persistent) characteristics (i.e., responses). One common and central purpose of current PUF technologies is to enable hardware identification via circuit measurements. Several varieties of PUFs have been proposed since being introduced by Pappu in [1] and range from optical PUFs that analyze the speckle pattern resulting from shining a laser beam on a transparent PUF to silicon timing PUFs.

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? And perhaps most importantly: can a personal computer be used to build truly secure PUF based protocols?

In the *real world*, the main security issue with using PUFs for off-line hardware identification and authentication is an attackers ability to simply virtualize the *useful* part of the PUF (i.e., a simple replay attack after observing one successful authentication of the legitimate PUF). In offline systems where PUFs are treated as black-box functions, preventing such replay attacks is a non-issue, since it is essentially impossible. Whereas, in the context of PUFs that are intrinsically involved in computation, preventing these replay attacks is both imperative and possible, as we show (with off-the-shelf hardware) later in this paper. It is imperative because not doing so potentially prompts some serious threats.

Contributions:

Our main contribution is a preliminary investigation into the questions posed above. We 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 attack described above and enables implict hardware identification without requiring peripheral PUF devices.

Furthermore, we investigate several system characteristics and present initial experimental results answering 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, etc.?. Finally, we briefly explain how these can provide better software protection and continuous authentication.

1.1 Related Work

Gassend, *et al.* [3] were the first to propose the use of silicon technology as PUFs citing the varying timing behavior of chips. The concept of using *intrinsic* PUFs for software protection on embedded systems was proposed by Guajardo, *et al.* [4] and by Simpson *et al.* [5]. However, these proposals depend on SRAM PUFs [6], making them vulnerable to read-out and replay attacks. Atallah, *et al.* [7] proposed using PUFs for software protection by intertwining PUF responses with software functionality. However, this work depends on trusted hardware (for initialization) and is not extendible to software that does not have non-algebraic functionality.

2. IP-PUFS: SOURCES OF UNPREDICTABILITY AND UNCLONABILITY

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 computers suitable for use as timing PUFs:

CPU Crystal Oscillator (CPU Clock): The two most interesting clocks for our purposes are the CPU clock and the timer interrupt clock. Given two crystals labeled with identical frequencies, it is unlikely that both oscillate at identical rates. This is due to several factors, such as crystal cut, impurities, and age. When this occurs with the CPU clock it has several effects – the actual time (in *picoseconds*) for the same instruction to be executed on two identical (in specification) CPUs is likely to be different – even though the instructions require the same number of clock cycles. This is confirmed by observing the sometimes varying *bogomips* values for processors from the same batch and family.

Timer Interrupt Clock: Differences in the oscillating frequency of the timer interrupt clock cause different definitions of a time quantum (i.e., for process schedulers performing round-robin scheduling) across multiple identical systems. A different number of clock cycles allocated to *one time quantum* results in a different number of scheduled instructions per quantum.

Memory: Assuming that our challenges are loaded in the processor cache (this is reasonable since the cache can always be flushed and filled as required), the time taken to load an instruction or data from the processor cache to register and vice-versa vary in the order of *picoseconds* from one system to another (even within the same family and architecture). This results in different latencies for load and store instructions across identical hardware.

3. EXPERIMENTS

In order to assess the practicality of our approach in the context of off-the-shelf computer systems, and also to understand the degree of unpredictability of time taken to execute the same set of instructions on a set of identical computers in controlled operating environments, we conducted a series of experiments on five identical machines with components (processors, PSU's, and RAM) that were from the same batch and family. The specifications of all the systems 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. We now describe our experimental setup and present our initial findings.

3.1 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 not dependent on the vagaries of the scheduler. To this end, all process signals and interrupts were blocked and the scheduler could not preempt the benchmarking program before completion.

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. Our timing measurements were obtained by reading the TSC (Time Stamp Counter) register – giving us a granularity of *nanoseconds* for each experiment. We argue that the timing inaccuracies of the TSC register are not harmful to our measurements since we only use these values to classify identical systems, and not actually harness these values for computation. However, we believe, a more accurate measure of time will be required to build applications that attempt to bind themselves to the PUFs as described in section 4.

Experiments were repeated 15000 times over multiple sessions to gather training data. Finally, a test sample was collected and a classifier (based on information gathered from the training data) was used to identify systems from the test data.

3.2 Results

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: Part of our results are illustrated in Fig. 1. Due to space restrictions, we present only a very brief analysis of our results. From the mean comparison chart we were correctly able to classify systems 3-515, 3-514, and 3-517, giving us a 60% success rate, with a confidence level of 90%. Note: 3-517 and one data point of 3-516 were all classified as the same system by our classifier.

Challenge-Response Variation: Certain instructions require more clock cycles or computations from different components of the computing device (e.g., any floating point operation inherits the timing characteristics and delays of the floating point unit). We were able to confirm through our experiments that even the same arithmetic operations, when using varying inputs, had slightly different execution times on the same system. This allows us to use any available

[1] *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.

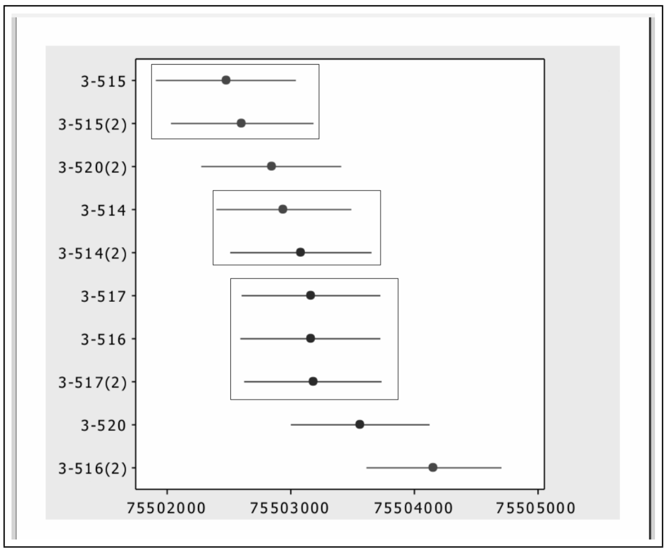

Figure 1: Intra Architecture Classifier Results. Here, the X-Axis denotes time taken (in nanoseconds) on average for 10,000,000 instructions, and the Y-Axis denotes the machine name (either 3-514, 3-515, 3-516, 3-517, or 3-520), and day of testing (either `null` or 2) (i.e., [Machine(Day)]).

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 specifications. We were able to successfully classify these systems correctly 100% of the time with a confidence level of 99.9%.

4. APPLICATIONS

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. In this section, we highlight the reasons and approaches through which the use of IP-PUFs are more likely to lead us to a more convincing solution.

Continuous Device Authentication and Software Protection: 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 part of the device that is actually challenged). Using IP-PUFs as described above, however, 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 pirate software by unhooking its functionality from the PUF – primarily due to the fact that debugging tools do not help the adversary (owing to the fact that they change the timing characteristics of the program being analyzed).

5. FUTURE WORK AND CONCLUSIONS

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 presented the preliminary results of our attempt to use of-the-shelf computers as PUFs. Based on timing characteristics alone, we achieved a 60% success rate in identifying computers with exactly identical hardware (from the same families) and architectures. IP-PUFs are already widespread, easily available, and easy to measure (via benchmarking suites) without the need for additional hardware. We also make the case that IP-PUFs are more useful for offline continuous device authentication and software protection owing to their ability to intertwine their behavior with software functionality. Further, IP-PUFs raise the bar for an attacker by negating the usefulness of virtual machines and debugging tools.

6. ACKNOWLEDGEMENTS

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.

7. REFERENCES

[1] R.S. Pappu, "Physical One Way Functions", *Ph.D Thesis, Massachusetts Institute of Technology,* 2001.

[2] P. Tuyls and B. Skoric, "Strong Authentication with Physical Unclonable Functions", *Security, Privacy, and Trust in Modern Data Management,* 2007.

[3] B. Gassend, "Physical Random Functions", *M.Sc Thesis, Massachusetts Institute of Technology,* 2003.

[4] J. Guajardo, S.S. Kumar, G.J. Schrijen, and P. Tuyls, "FPGA Intrinsic PUFs and Their Use for IP Protection", *Workshop on Cryptographic Hardware and Embedded Systems (CHES),* 2007.

[5] E. Simpson and P. Schaumont, "Offline HW/SW Authentication for Reconfigurable Platforms", *Workshop on Cryptographic Hardware and Embedded Systems (CHES),* 2006.

[6] C. Bohm and M. Hofer, "Using SRAMs as Physical Unclonable Functions", *Austrochip - Workshop on Microelectronics,* 2009.

[7] M. Atallah, E. Bryant, J.T. Korb, and J.R. Rice, "Binding Software to a Specific Native Hardware in a VM Environment: The PUF Challenge", *ACM Workshop on Virtual Machine Security (VMSec),* 2008.