A BENIGN HARDWARE TROJAN ON FPGA-BASED EMBEDDED SYSTEMS

Jason X. Zheng, Ethan Chen, Miodrag Potkonjak

Department of Computer Science
University of California, Los Angeles (UCLA)
Los Angeles, USA
email:{jxzheng,ethanc,miodrag}@cs.ucla.edu

ABSTRACT

In this paper we present the use of Benign Hardware Trojans (BHT) as a security measure for an embedded system with a software component and a hardware execution environment. Based on delay logic, process variation, and selective transistor aging, the BHT can be incorporated into an embedded system for the software and the hardware components to authenticate each other before functional execution. We will demonstrate an implementation of such a BHT within an embedded system on a Xilinx Spartan-6 FPGA platform. Using the same platform we will also show that the BHT security measurement has a low to modest amount of performance overhead basing on the test results from a variety of synthetic and real world benchmarks.

1. INTRODUCTION

Hardware Trojan Horses (HTH) are hidden structural and functional alterations of an integrated circuits in such a way that the integrity or the privacy of the stored information are compromised. A favorite fictional example of the authors is an HTH-infested Ethernet interface card that can secretly transmit sensitive information to remote servers.

In this paper we discuss a less malicious type of HTH, whose intent is not to penetrate security walls, but to ward off security attacks on embedded systems, such as code-injection, reverse-engineering, and cloning. To be clear, the embedded system under consideration consists of a software component and a hardware component, where the software executes. The Benign Hardware Trojan (BHT), when embedded in the hardware component of the system, acts as a gate keeper to guarantee the following properties:

1. That the software binary S_1 executing on the hardware platform H_1 is authorized to execute on H_1.
2. That the hardware platform H_1 is an authorized execution target of the software binary S_1.
3. That violating either of the previous two properties will cause the system to crash or stall.

1.1. Shared Uniqueness

Intuitively, the BHT is analogous to the unique genetic markings of a biological host waiting for a transplant. Only when the host (H_1) and the transplant (S_1) contain matching genetic markings can the transplant proceed. A key concept of the BHT implementation, therefore, is the embodiment of a shared uniqueness between H_1 and S_1, i.e. the software binary shall be tailored to only execute properly when the hardware manifests the shared uniqueness.

We propose to provide this uniqueness by exploiting process variation and aging effects from negative-bias temperature instability (NBTI) ([1]). Using delay-sensitive logic, whose output behaviors are dependent on not only the structure of the logic circuits but also the relative speeds of competing logic paths ([2]), the minuscule intra- and inter-die delay variations can be translated to control signals whose outputs cannot be predicted or preset prior to manufacturing. In addition to delay variations that are introduced by the process variation, selective aging can also be used to increase the delay disparities in the BHT.

Since the uniqueness of the hardware platform is a manifestation of a random physical process, reproducing it is extremely difficult, and the difficulty of reproduction increases exponentially as the size of the BHT logic increases.

Figure 1 demonstrates the feasibility of extracting unique silicon signatures based on process variation. In this figure, two structurally identical FPGAs (Xilinx Spartan-6) are configured using the same logic configuration, consisting of an array of 64 delay-sensitive arbiters. The outputs of the arbiters are dependent on the relative logic speeds of two competing paths, i.e. the output is zero if one path wins, and one if the other paths wins. The y-axis shows the accumulated output values of each arbiter over 1024 iterations. Roughly 30% of the arbiter outputs are different between the two "identical" FPGAs.

1.2. Multi-Core Processor Example

In this section, we show a practical and intuitive example of incorporating the hardware signature with the software com-

978-1-4673-2256-0/12/$31.00

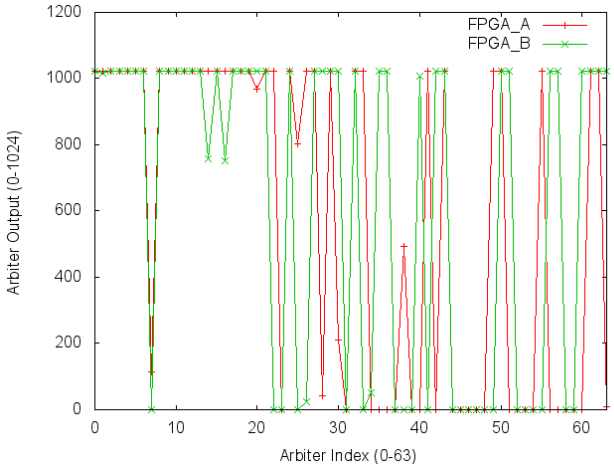

Fig. 1. Comparison of arbiter outputs from two FPGAs

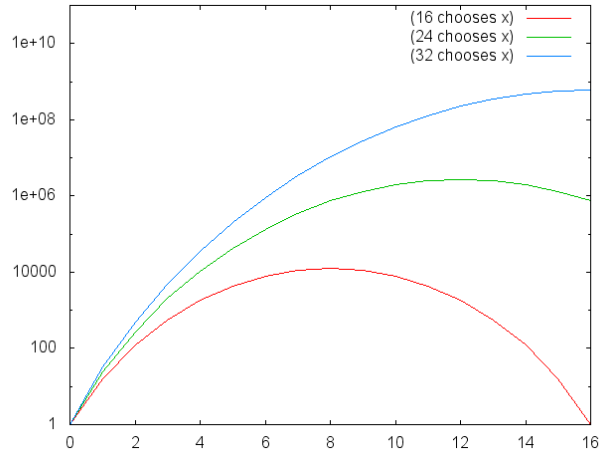

Fig. 2. Choices for disabling cores when total number of cores is 16, 24, or 32

pilation process, which allows the software and the hardware to authorize each other during execution. This example is a system with a tiled multi-core processor (e.g. a tiled multi-core DSP processor) with as many as hundreds of processing cores. The highly optimized software implements a massively parallel algorithm with a *static* schedule of the processor cores.

We then use process variation to randomly choose N cores to be logically disabled by the BHT in each die. When the software is compiled, the states of the disabled cores are passed to the compiler to schedule the execution in such a way that the real computation only occurs on the enabled cores. The compiler can still schedule operations on the disabled cores, as long as the results do not contribute to the final answer.

We now analyze several possible reverse engineering or cloning attacks at the system and the defense options against them.

Let us assume that both the software and the hardware implementations are highly sophisticated, and therefore only low-level reverse engineering is possible. For the software component, it is possible to dis-assemble the binary code, but nearly impossible to obtain a high-level description of the software design. For the multi-core processor, we assume that attacker can reverse-engineer the chip layout and routing or the configuration bitstream for FPGA designs to generate a low-level netlist. However, reverting the netlist to a behavioral description of the processor is also nearly impossible.

The first attack option is to attempt to clone the hardware system in such a way that all the processing cores are enabled after manufacturing. This attack shall be known as the functional clone attack. The functional clone is very difficult because a low-level cloning typically requires the clone to be as close to the originally as possible. If the attacker makes a clone that's structurally identical to the original, processing variation will guarantee that some cores will always be disabled by the BHT after manufacturing. Furthermore, the software binary can also include checks that verify the states of the cores that are meant to be disabled.

A second option is to manufacture a massive number of the hardware clones and hope to find one that will work with the cloned software code. This attack shall be known as the mining attack. The mining attack is prohibitively expensive, as the number of core choices grows binomially with the number of total number of cores. Figure 2 shows the growth of choices on a log scale with 16, 24, and 32 total number of cores. When half of the cores are disabled by BHT, the growth of total choices is close to exponential growth.

A third option is to perform statistical analysis of the software binary's scheduling behavior to infer which processor cores are used and which ones are not. The attacker can then clone the hardware, map out the usable and unusable cores the cloned hardware, and then patch up the static schedule in the software binary to use only the usable cores. This attack shall be known as the static analysis attack. The static analysis attack can also be guarded against if the compiler obfuscates the binary by scheduling both the functional and broken cores to active duty, but discard the outputs of the broken cores.

The final attack, dynamic analysis attack, is an extension to the static analysis attack. In a dynamic analysis attack, the attacker can write a special test software to identify the disabled processing cores in the original processor, and use NBTI aging to make the clones behave like the original. The defense against the dynamic analysis attack, incidentally, is also NBTI aging. The BHT can be designed in such a way that its behavior and state is software-dependent. When the matched software executes on the hardware platform, it

enforces an aging pattern which modifies the output of the BHT. Without understanding the aging profile applied by the original software, the special test software cannot learn the true state of the BHT.

2. RELATED WORKS

In this section we briefly survey directly related efforts, ranging from enabling technologies and techniques such as process variation, device aging, and hardware trojans to already existing techniques for the protection of hardware and software.

Process variation is widely acknowledged as one of the most important challenges in modern designs ([3]). Synthesis for FPGA considering process variation is an important research task that has been well addressed ([4], [5]). Negative-Bias Temperature Instability (NBTI) is an intrinsic property of CMOS technologies, where each transistor under negative V_{gs} bias is subject to stress. A main ramification of NBTI is a significant speed degradation of the pertinent gates ([1], [6]). Employment of device aging has been proposed for software metering ([7]), hardware-based cryptography ([8]), and prevention of reverse engineering ([2]).

Hardware trojan has attracted a great deal of attention in the last five years ([9], [10]). Some of the more recent detection techniques do not only detect the presence of hardware trojans, but also enable their diagnosis, masking, or elimination ([11], [12], [13], [14], [15], [16]). There has been a number of efforts to design small and/or highly damaging hardware trojans ([17], [18], [19]). However, while all these efforts created trojan horses with the intention to demonstrate their practical importance and/or to facilitate development of the detection techniques, in our case the purpose of the trojan horse is to directly enable the creation of new software and hardware protection techniques.

Several techniques for the protection of hardware intellectual property have been proposed. They can be classified into active and passive techniques. Active techniques have a high hardware overhead for remote activation and disabling ([20]). Passive metering techniques have low cost overhead but they do not provide enforcement mechanisms ([21]). Dabiri et al. proposed the addition of a high area and energy specialty circuitry for the remote chip enabling and disabling and the software usage metering ([7]).

To the best of our knowledge the new approach is the first software control that can used on both programmable processor and FPGAs. It does not require any additional circuitry when the platform is FPGA because all the required device aging can be done using an FPGA configuration, which is consequently replaced with actual functionality.

3. IMPLEMENTATION APPROACH

The multi-core processor case in Section 1.2 serves as an intuitive example of BHT, and it would have been an impressive demonstration if implemented. However, due to the limited reconfigurable logic resources available at our disposal, we choose to demonstrate the BHT with a simpler form of commodity resources in an embedded system. We would like to show that BHT can be used to control the disabling of the general purpose registers (GPRs) on an embedded microprocessor with a low performance overhead. With the latency of a single cycle, GPRs are the processor's fastest and most frequently used data storage unit. Similar to the multi-core processor example, there are usually more than enough GPRs on a modern processor architecture, and any typical modern compiler can handle register scheduling with an arbitrary set of GPRs.

Following the example of the multi-core processors, we incorporate the BHT into the processor to control the disabling of GPRs. The modified processor is synthesized, placed, and routed on an FPGA platform to run. The outputs of the BHT control path is characterized dynamically when the processor is running. With a BHT array directly controlling which GPRs are enabled, both the functional clone attack and the mining attack are deterred.

On the compiler side, the compiler is modified such that the knowledge of which GPRs are not available is to be passed on at the compile time. The compiler then generates a binary code that only use the GPRs that are available. Finally, the compiled code is executed on the physical processor on the FPGA board for benchmarking.

To guard against the static and the dynamic analysis attacks, the compiler requires further modification to its scheduling algorithm and additional NBTI characterization on the selected target device to show the effects of NBTI on the BHTs (a similar characterization was performed on Virtex-5 FPGAs in [2]). These two attacks will be addressed in a future implementation.

3.1. Hardware Implementation

We choose to implement the BHT on an open-sourced processor. The OpenRISC OR1200 is a five-stage, 32-bit general-purpose processor. There are 32 general-purpose registers in the architecture, and most of which can be masked without functional impact to the processor (exceptions are covered in Section 3.2). We choose the OpenRISC platform because the complete set of source code is available (in Verilog format), which is critical to modifying the processor design.

The BHT is implemented as delay-logic arbiters as described in [2], whose outputs enables or disables writing to particular GPRs (Figure 3). Each arbiter output corresponds to a particular GPR number. When a disabled GPR is decoded as the destination register for an instruction, the out-

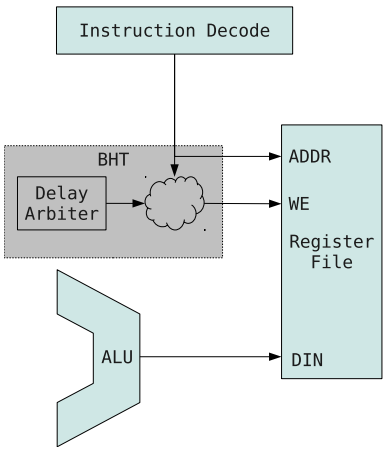

Fig. 3. BHT masking selected GPR writes

Table 1. OR1200 Resource Usage

Resource	Use Count	Use Percentage
D-Flipflop	5718	10%
LUTs	10918	40%
Slices	3661	53%
BRAMs	87	37.5%
DSP48A1s	4	6%

put of the BHT masks the write, thus effectively disabling the register.

The Digilent Atlys board from the Xilinx University Program is used as the target platform, which has a 45nm-process Spartan-6 FPGA and 128 megabytes of DDR2 memory. The OR1200 processor is implemented with a full suite of on-chip peripherals, including a DDR2 memory controller, Ethernet controller, serial communication, hardware-based multiplier, divider, and floating-point unit. The processor and its peripherals takes up roughly 50% of the available logic resources, and is clocked at 50MHz. Table 1 shows a summary of the resources used in the FPGA, and Figure 4 shows the final layout of the design.

3.2. Software Implementation

As an embedded platform, OpenRISC offers several options for building and running code. OpenRISC utilizes the common GNU toolchain for 32-bit C/C++ support, including GCC, binutils, and GDB. However, library support is provided by either Newlib, uClibc, or the low-level System-on-Chip (SoC) platform libraries. Newlib provides libc support for stand-alone bare-metal code, while uClibc provides libc support for Linux. The much more limited SoC platform libraries can also be utilized to run bare-metal code, although they provide only a small subset of the functionality available from the libc implementations. Although libc libraries like Newlib and uClibc are available, much functionality is still provided through OpenRISC-specific extensions, and not the standard functions. In addition to the GNU toolchain, the OpenRISC platform provides additional debugging tools such as the adv_debug_bridge JTAG proxy and the or1ksim architectural simulator.

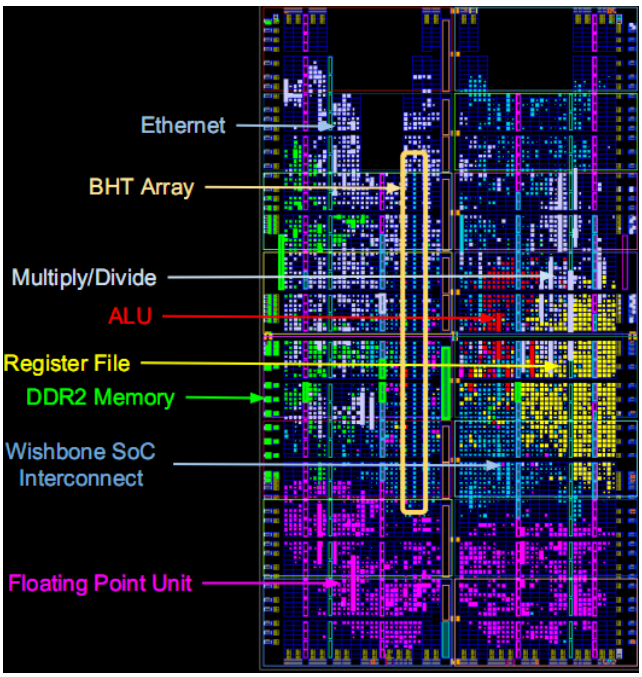

Fig. 4. Layout of the implemented OR1200 with BHT.

As we aim to evaluate the performance impact of reduced general purpose register availability, the software toolchain of the OpenRISC platform must be modified to allow arbitrary sets of registers to be disabled. We do this by extending the options available in the GCC OpenRISC target to modify register availability. This functionality is available as a command line option under the -mregistermask flag, taking a 32-bit value that represents the state of each register as indexed by the bit positions, where the least significant bit represents GRR0. Setting the bit makes the corresponding GPR unavailable to the compiler.

Although the 32-bit OpenRISC architecture defines 32 general purpose registers, several are reserved for special purposes by the GCC OpenRISC machine definition and cannot be disabled. They are as follows:

- R0, defined as the zero constant by the 32-bit OpenRISC architecture,
- R1, defined as the stack pointer by the GCC,
- R9, defined as the link pointer by the GCC,
- R10, defined as the Linux thread register by the GCC.

In addition to the GPR modifications in the GCC, we modified the or1ksim architectural simulator to simulate disabled registers for testing purposes. In much the same way that the -mregistermask GCC flag works, the CPU configuration option disable_regs was implemented in the modified or1ksim to selectively disable registers using a 32-bit integer mask. By inserting code into the simulator state machine generated, register checks were added to trigger a processor exception upon accessing disabled GPR. This allowed us to do preliminary characterization using the simulator's built-in cycle counter.

4. BENCHMARK RESULTS

Due to the fact the scheme relies on disabling registers, the performance impact must be considered. We have thus benchmarked the performance of the modified OpenRISC platform using a variety of programs, varying the numbers of disabled registers. As the OpenRISC platform is an embedded system with limited support for file access to storage, we relied primarily on self-contained embedded system benchmarks. Evaluations of register performance penalties were done using a small suite of benchmarks consisting of Dhrystone ([22]), CoreMark ([23]), a subset of MiBench ([24]), and zlib ([25]). Dhrystone and CoreMark are synthetic benchmarks that attempt to capture typical CPU workloads, while MiBench attempts to characterize CPU performance based on a variety of common operations, including FFT (Fast Fourier Transformation), CRC-32 (Circular Redundancy Checks), string searching, media decode and encode, and encryption.

Both CoreMark and Dhrystone are completely synthetic benchmarks, and not centered around real workloads, but instead seek to approximate real embedded workloads. Dhrystone has been an industry standard since the 1990s, and consists of a mix of assignments, pointer arithmetic, and procedure calls. CoreMark is created by the Embedded Microprocessor Benchmark Consortium (EEMBC) targeted at evaluating embedded systems. It consists of modules that evaluate linked list, matrix, and state machine performance, making it a more realistic simulation of common embedded workloads than Dhrystone.

MiBench was originally developed at the University of Michigan to evaluate embedded system performance as CoreMark was previously not freely available. It consists of 35 individual modules that are based on a variety of real-world workloads. As most of the MiBench modules rely on external data to operate, we are only able to make use of a subset of the modules, including the basic math, bitcount, FFT, and string search modules. The basic math module is designed to test math operations that are not commonly supported by hardware, such as integer square root, cubic function solving, radian to degree conversion, etc. The bitcount module is designed to evaluate the bit manipulation performance of an embedded processor, counting bits using a variety of methods. The FFT module performs a floating point Fast Fourier Transform on a polynomial function with pseudo-randomly generated amplitude and frequency components. Finally, the string search module evaluates performance using a fast string search to find strings in a large array of input phrases.

The zlib benchmark is naturally designed as a real-world test of compression performance, as the DEFLATE algorithm used in zlib is used very frequently in real software. The benchmark itself is very simple and centers around initializing large chunks of random data, followed by compressing those chunks using calls to deflate functions in zlib. Since the OpenRISC platform does not allow file access, input data is randomly generated and the resulting compressed data is merely thrown away, making this a test purely of compression speed without file I/O to slow things down.

4.1. Measurements Methodology

To measure the performance using these various benchmarks, we used extensions present in the OpenRISC Newlib implementation. These extensions provide access to the CPU timer, enabling accurate timing, as Newlib does not provide a functional implementation of standard libc timer code for the OpenRISC port. Timer accuracy is set to 1000Hz for all measurements, making the ticks measured equivalent to time in milliseconds. Each of the benchmarks measures elapsed time in number of ticks.

To see the impact of available registers on benchmark performance, the number of available GPRs are varied with the following numbers: 32, 28, 24, 22, 20, 18, and 16. With each variation, the benchmark is recompiled with a corresponding register mask. Each test is repeated three times, and the lowest number is used as the result. The first run is typically slower due to cache misses.

As fewer registers are available, the processor is forced to visit the memory system more frequently. A special test is run to evaluate the cache dependence of the performance penalty. This test is only run with CoreMark, where the entire test is repeated with the cache disabled.

4.2. Performance Penalty

The raw benchmark results are tabulated in Table 2. Using the results from perspective 32-GPR runs, the benchmark numbers are normalized to a base number of 100, and the results are graphed in Figure 5.

Excluding the cache-disabled CoreMark runs, the worst case performance penalty is 8% for zlib with 22 GPRs. Most of the benchmarks show a monotonic increase in run time with the decrease of available GPRs. The Dhrystone results is an interesting outlier, where we observed the worst-case

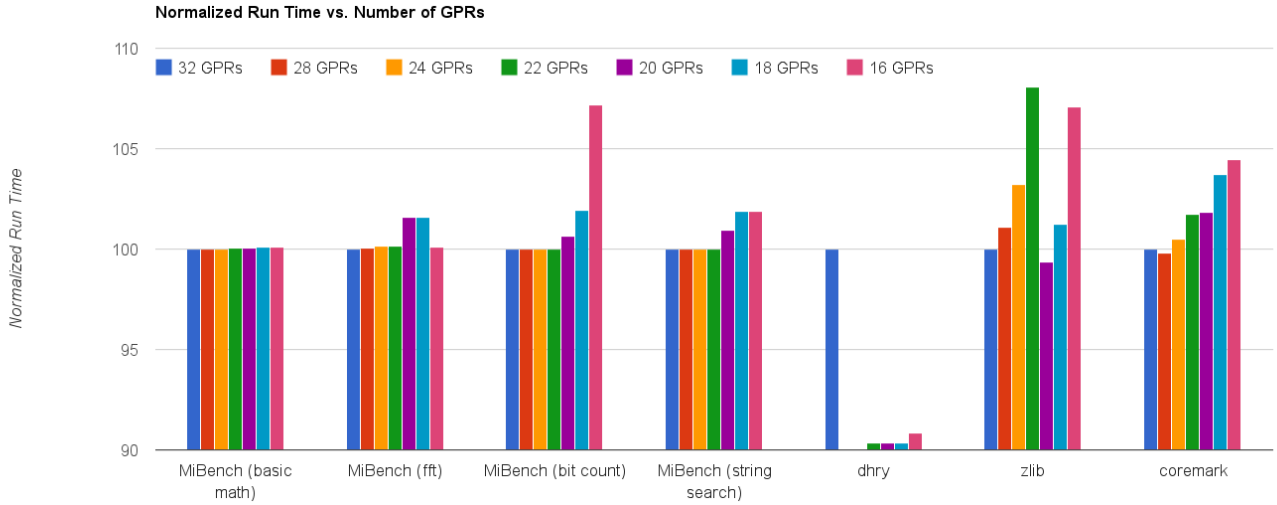

Fig. 5. Performance penalties from fewer GPRs

Table 2. Raw Benchmark Run Times (1000Hz ticks)

Benchmark	32 GPRs	28 GPRs	24 GPRs	22 GPRs	20 GPRs	18 GPRs	16 GPRs
MiBench.basicmath	39640	39641	39645	39659	39667	39675	39680
MiBench.fft	6570	6574	6580	6580	6674	6675	6578
MiBench.bitcnt	24325	24325	24325	24325	24483	24800	26068
MiBench.stringsearch	106	106	106	106	107	108	108
Dhrystone	4032	1633	1642	3641	3645	3642	3662
CoreMark	1563	1560	1571	1590	1592	1621	1633
CoreMark.no_cache	2102	2113	2130	2145	2162	2210	2522
zlib	67060	67788	69209	72458	66641	67884	71794

performance when all 32 GPRs are available. We are not completely surprised by this counter-intuitive behavior, as the Dhrystone benchmark performance has been shown to be highly dependent on compiler optimization ([22]).

A similar anomaly can be seen with the zlib results. The run time monotonically increases until the number of available GPRs is reduced to 20, where the run time drops to lower than the 32-GPR run, and then increases monotonically again. We suspect that this is also an artifact of the compiler optimization choices.

The basic math and FFT module results from MiBench show very little run time impact as a result of reduced general purpose register availability. This is likely due to the fact that basic math performance is limited by arithmetic hardware, rather than register availability, while FFT largely performs floating point operations that do not use the general purpose register set, and is limited by the Floating Point Unit's (FPU) latency.

The impact of constraining the number of GPRs in the absence of cache can be observed in Figure 6. The absence of cache compounds the performance penalty at every variation of GPR numbers. The effect is especially high when only 16 GPRs are available, resulting in a 20% increase in run time.

5. CONCLUSION

We have presented a novel use of Benign Hardware Trojan as a security measure when combined with delay logic, process variation, and NBTI aging. We have shown through example that the BHT can be very resilient against reverse-engineering and cloning attacks. Furthermore, we have demonstrated a successful implementation of an embedded system incorporated with BHT. Finally, we have shown that adding BHT as a security measure only brings about a modest amount of performance penalty when tested against a

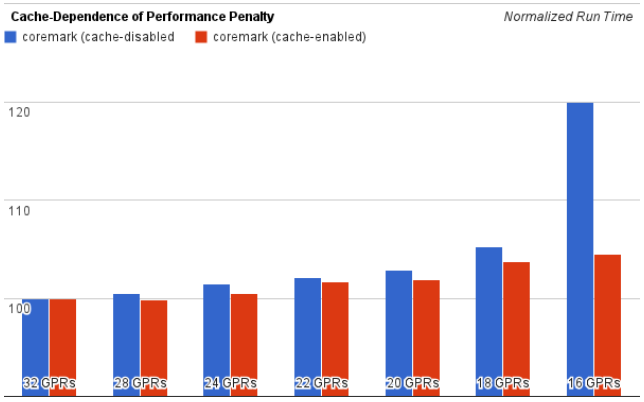

Fig. 6. Performance penalty and cache

suite of benchmarks.

6. ACKNOWLEDGMENT

This work was supported in part by the NSF under awards CNS-0958369, CNS-1059435, and CCF-0926127, and is based upon research performed in collaborative facilities renovated with funds from the National Science Foundation under Grant No. 0963183, an award funded under the American Recovery and Reinvestment Act of 2009 (ARRA).

7. REFERENCES

[1] D. K. Schroder, "Negative bias temperature instability: What do we understand?" *Microelectronics Reliability*, vol. 47, no. 6, pp. 841–852, 2007.

[2] J. X. Zheng and M. Potkonjak, "Securing netlist-level FPGA design through exploiting process variation and degradation," in *FPGA*, Feb. 2012, pp. 129–138.

[3] S. Borkar *et al.*, "Parameter variations and impact on circuits and microarchitecture," in *DAC*. New York, NY, USA: ACM, 2003, pp. 338–342.

[4] L. Cheng *et al.*, "FPGA performance optimization via chipwise placement considering process variations," in *FPL*, Aug. 2006, pp. 1 –6.

[5] A. Kumar and M. Anis, "FPGA design for timing yield under process variations," *IEEE Transactions on VLSI Systems*, vol. 18, no. 3, pp. 423–435, March 2010.

[6] B. Paul *et al.*, "Impact of NBTI on the temporal performance degradation of digital circuits," *Electron Device Letters, IEEE*, vol. 26, no. 8, pp. 560–562, Aug. 2005.

[7] F. Dabiri and M. Potkonjak, "Hardware aging-based software metering," in *DATE*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 460–465.

[8] S. Meguerdichian and M. Potkonjak, "Device aging-based physically unclonable functions," in *DAC*. New York, NY, USA: ACM, 2011, pp. 288–289.

[9] D. Agrawal *et al.*, "Trojan detection using IC fingerprinting," in *IEEE Security*, May 2007, pp. 296–310.

[10] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *Design Test of Computers, IEEE*, vol. 27, no. 1, pp. 10–25, Jan. 2010.

[11] S. Wei, S. Meguerdichian, and M. Potkonjak, "Malicious circuitry detection using thermal conditioning," *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 3, pp. 1136–1145, Sept. 2011.

[12] S. Wei and M. Potkonjak, "Scalable consistency-based hardware trojan detection and diagnosis," in *NSS*, Sept. 2011, pp. 176–183.

[13] M. Potkonjak, A. Nahapetian, *et al.*, "Hardware trojan horse detection using gate-level characterization," in *DAC*, 2009, pp. 688–693.

[14] S. Wei, K. Li, *et al.*, "Hardware trojan horse benchmark via optimal creation and placement of malicious circuitry," in *DAC*, 2012, pp. 90–95.

[15] S. Wei and M. Potkonjak, "Wireless security techniques for coordinated manufacturing and on-line hardware trojan detection," in *WISEC*, 2012, pp. 161–172.

[16] S. Wei and M. Potkonjak, "Scalable hardware trojan diagnosis," *IEEE Transactions on VLSI Systems*, vol. 20, no. 6, pp. 1049–1057, 2012.

[17] S. King *et al.*, "Designing and implementing malicious hardware," in *LEET*, Apr. 2008, pp. 1 –8.

[18] L. Lin *et al.*, "Trojan side-channels: Lightweight hardware trojans through side-channel engineering," in *CHES*, C. Clavier and K. Gaj, Eds. Springer Berlin / Heidelberg, 2009, vol. 5747, pp. 382–395.

[19] J.-F. Gallais *et al.*, "Hardware trojans for inducing or amplifying side-channel leakage of cryptographic software," in *Trusted Systems*. Springer Berlin / Heidelberg, 2011, vol. 6802, pp. 253–270.

[20] Y. Alkabani *et al.*, "Remote activation of ics for piracy prevention and digital right management," in *ICCAD*. Piscataway, NJ, USA: IEEE Press, 2007, pp. 674–677.

[21] Y. Alkabani, F. Koushanfar, *et al.*, "Trusted integrated circuits: A nondestructive hidden characteristics extraction approach," in *Information Hiding*. Springer Berlin / Heidelberg, 2008, vol. 5284, pp. 102–117.

[22] R. Weicker, "An overview of common benchmarks," *Computer*, vol. 23, no. 12, pp. 65–75, Dec. 1990.

[23] "Coremark, an EEMBC benchmark," 2012. [Online]. Available: http://coremark.org

[24] M. R. Guthaus *et al.*, "MiBench: A free, commercially representative embedded benchmark suite," in *IISWC*, Dec. 2001, pp. 3–14.

[25] P. Deutsch and J.-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3," RFC 1950 (Informational), Internet Engineering Task Force, May 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1950.txt