Detecting/Preventing Information Leakage on the Memory Bus due to Malicious Hardware [1]

Abhishek Das, Gokhan Memik, Joseph Zambreno[+] and Alok Choudhary

Electrical Engineering and Computer Science Department
Northwestern University
Evanston, IL USA

[+]Electrical and Computer Engineering Department
Iowa State University
Ames, IA USA

***Abstract*—An increasing concern amongst designers and integrators of military and defense-related systems is the underlying security of the individual microprocessor components that make up these systems. Malicious circuitry can be inserted and hidden at several stages of the design process through the use of third-party Intellectual Property (IP), design tools, and manufacturing facilities. Such hardware Trojan circuitry has been shown to be capable of shutting down the main processor after a random number of cycles, broadcasting sensitive information over the bus, and bypassing software authentication mechanisms. In this work, we propose an architecture that can prevent information leakage due to such malicious hardware. Our technique is based on guaranteeing certain behavior in the memory system, which will be checked at an external guardian core that "approves" each memory request. By sitting between off-chip memory and the main core, the guardian core can monitor bus activity and verify the compiler-defined correctness of all memory writes. Experimental results on a conventional x86 platform demonstrate that application binaries can be statically re-instrumented to coordinate with the guardian core to monitor off-chip access, resulting in less than 60% overhead for the majority of the studied benchmarks.**

1. Introduction

The high economic cost of fabrication is pushing semiconductor companies to move their fabrication technology to overseas locations, or to go completely fab-less and outsource fabrication to other companies. Moreover there is an increasing trend of using third-party Intellectual Property (IP) cores and tools. These developments suggest that modern microchips are designed and fabricated in relatively less trusted environments. A security breach in the fabrication and masking stages can lead to what is called a hidden circuit or a Trojan hardware attack. Recent research suggests four main threats from malicious hardware [1]: the chip could be reverse engineered, with sensitive IP being leaked out to competitors; the chip could be counterfeited by simply manufacturing additional copies; the chip's life span could be greatly shortened through manipulation of subtle electrical characteristics; and malicious hardware, which can be used for various purposes, could be inserted.

The insertion of malicious hardware is potentially the most troublesome threat, given that this hardware could be used in order to break encryption, broadcast sensitive information, circumvent authentication mechanisms, or disable or modify intended software execution. This added functionality might be small in terms of design complexity. As the malicious hardware will fall outside the predetermined range of testing parameters, it is thought to be inherently undetectable via traditional visual and functional chip verification methodologies.

In this paper, we propose a system for detecting information leakage attacks. The system is based upon the concept of a guardian core in order to protect against Trojan circuits that aim to leak confidential data. This work is motivated by the observation that both information leakage and unexpected execution events can only take place given surreptitious processor writes to memory. Malicious memory reads can be essentially ignored, as the flow of information stays internal to the chip until the subsequent memory write operation. Our guardian core acts as a gatekeeper for the memory write operations executed by the main core. Specifically, the program binary is instrumented to generate two memory write operations for each of the writes in the original code. The guardian core then checks whether the memory stream exhibits the predetermined properties of the instrumented binary. Any deviation will be detected and stopped at the guardian core preventing the information leakage. Overall, by sitting between off-chip memory and the main core, the guardian core can monitor bus activity and verify the compiler-defined correctness of all memory writes.

There are several advantages of our scheme. First, the underlying implementation can be transparent to the application developer. Through the development of an x86 static binary instrumentation infrastructure, we demonstrate the applicability of our approach to legacy code. The inclusion of reconfigurable logic fabric can make the underlying guardian core implementation transparent to the hardware designer as well. Experimental results show that the overhead of the technique is dependent on the memory access behavior of the application. For compute intensive applications the overhead is as low as 1.7%. The average overhead for the compute intensive benchmarks is 31.2%. For highly data-intensive applications, on the other hand, the overhead can be over 100%; reaching up to 396% for one of the target applications.

The remainder of this paper is organized as follows. Section 2 provides a brief background on possible Trojan hardware attacks. In Section 3, we present our information leakage detection framework. A detailed description of the scheme and various design challenges are dealt in Section 4. Section 5 illustrates the experimental results obtained from our scheme, and Section 6 discusses related work in this area. Finally, the paper is concluded in Section 7 with a brief summary of the work.

[1] This work was supported in part by NSF grants CCF-0916746, CCF-0747201, CNS-0830927, CNS-0551639, IIS-0536994, HECURA CCF-0621443, OCI 0956311, OCI 0724599, and SDCI OCI-0724599.

2. Malicious Circuits: Attack Model

Mobile devices or personal computers could be built out of malicious chips even when the software is subjected to the highest standards of security in its design and operation, i.e., the software is completely encrypted in main memory and is only decrypted at the time of execution in the processor. A hidden circuit inside the processor can thwart this approach by seizing execution control at an opportune moment and writing out decryption keys onto peripherals. An even more rudimentary attack is to simply halt the processor at the worst possible moment, at a critical or random time long after the processor has been in use [2]. Trojan circuits can also be set up to scan for electromagnetic signals so that a processor can be shutdown when provided the right external cue. Another possible attack is to use such circuits to facilitate reverse engineering of system design. Attacks using malicious hardware usually involve a much larger investment than software alone attacks; hence they often target higher-value systems. Therefore, the problem we have described is of particular interest to military, government applications, as well as to commercial entities concerned with guarding their intellectual property.

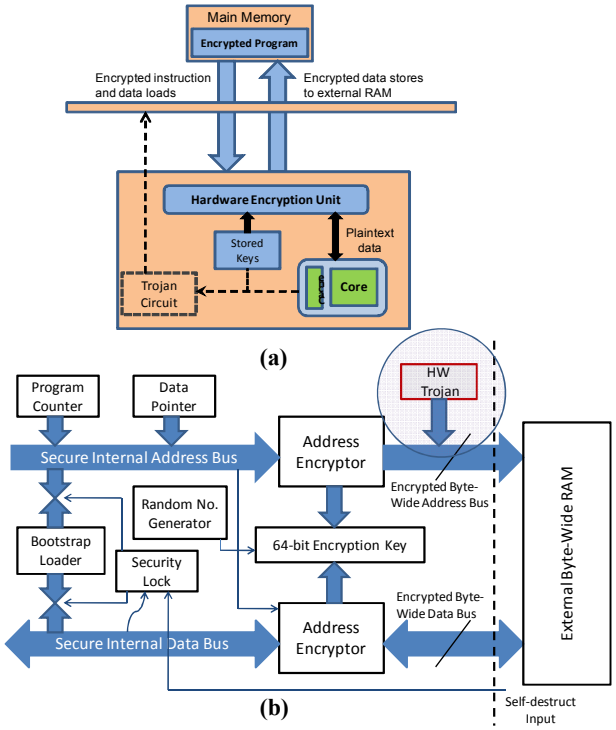

Figure 1. (a) A hardware Trojan circuit leaking confidential information and encryption keys, and, (b) an address bus corruption in the DS5002FP [3] secure processor component

Leakage attacks can be caused by leaking out secret information to the memory bus, where an adversary can easily monitor the bus activity and decipher the secret [4]. Figure 1a describes a high level view of information leakage whereby the stored encryption keys are leaked out on the system bus. Figure 1b shows such tailor-made attacks can easily fit within a commercially available secure processor (DS5002FP) manufactured by Dallas Semiconductors [3]. In both cases, the hardware encryption unit can be bypassed and the keys can be broadcasted onto the external memory bus. Previous works show that testing techniques like scan chains can be used to attack stored information in registers [5]. Our primary goal is to detect a leakage of information through any possible Trojan circuitry.

3. Information Leakage Detection

In most cases, information leakage due to hidden on-chip malicious circuits takes place due to illegal writes to the main memory. Data fetches and reads are not a concern, since even if some confidential information is read from the memory it cannot be leaked to the external world unless it is leaked out on to the memory bus. Assuming there is no external data interface (e.g., data/network ports) on the chip itself other than the address and data bus, whatever goes out of the core must reside in the memory first. Thus, only stores to a memory location can be considered as a possible malicious operation by the processor; a Trojan circuit can leak information by exploiting such vulnerabilities. For example, there may be a simple control circuitry inside the processor that multiplexes the address on the address bus and inserts a new store operation in the execution flow. This store in itself can leak out some confidential information. Such an attack can be easily implemented by a sophisticated attacker. More importantly, it will be very difficult for a user to understand the existence of such circuits. Therefore, to prevent information leakage the end-users and/or manufacturers need to counter the attacks that target stores to memory bus.

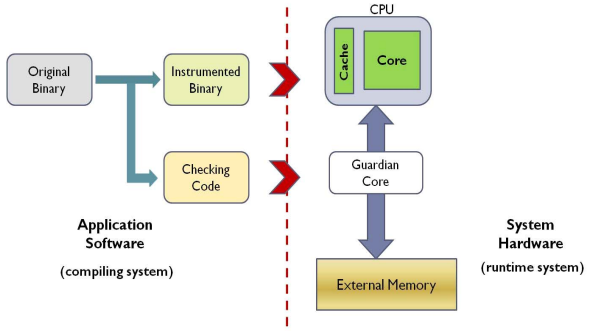

Figure 2. Overall hardware-software approach for securing information leakage.

The core idea behind our malicious leakage detection framework lies in being able to replicate the memory operations in an application. Specifically, for a stream of memory accesses (data reads or writes) in a given application, an equivalent stream can be generated which emulates the memory access behavior. In other words, the source code/binary can be modified to generate two simultaneous streams: the original one and a parallel one for which the addresses are generated using a secret key. If the "parallel" stream bears a one to one symmetric key mapping from the original stream, it will be possible to detect any alteration of the original address/data by comparing with the newly generated address/data. In other words, if for all memory operations in a program a matching memory operation is inserted in the same program, then changes in the original address stream can be detected by comparing with the second memory stream. Similarly, an insertion of any new memory operation will also be detected. Hence, any data that will be sent out by the Trojan circuitry can be detected and blocked. We refer to the newly generated stream as a shadow stream of the original memory operations. We demonstrate through our work that such a shadow stream can be generated by instrumenting the original binary.

Once a set of memory operations can be emulated, any secondary execution core sitting on the memory bus can perform a checking whether the shadow stream and the original stream

bear a predefined mapping or not. Using a specified set of keys and a mapping function, a mismatch in the address operands of the original memory instructions and the emulated (shadow) stream will infer that a possible information leakage attack has been launched by some malicious circuitry inside the processor.

Figure 2 illustrates the overall mechanism of our scheme, which consists of two main components. First, an application binary deployed by a software vendor can be modified by the end user to provide additional security features. Specifically, the end user can utilize a confidential key and instrument the application binary to generate a new stream of memory operations which can be incorporated in the original binary. Second, an external reprogrammable core sitting on the memory bus can be programmed by the user in such a way that it can check whether the mapping between the two memory streams holds or not. If there is a mismatch, this external core can raise a flag that an information leakage attack has taken place. We call such a core the ***guardian core (GC)***. The checking code for the reprogrammable core can be generated during the binary instrumentation stage. Note that we do not handle the case where there is collusion between the GC and the processor core, i.e., the case where the processor and the GC both have malicious hardware in them incorporated by the same attacker.

4. Detailed Approach

To guarantee such bus pattern, we instrument a shadow operation for every memory operation in the binary. In other words, our goal is to generate a shadow store operation for all the write operations in the original binary. If all store instructions in a processor can be confirmed, information leakage attacks can be detected. However, if we only replicate the store operations, we cannot guarantee that the shadow store data and the original store data will be evicted at the same time. Therefore, we have to also replicate memory read operations to guarantee eviction of the stores (details will be provided in the following subsection). For example, let us say an application software contains instructions with memory access of the form `store X,(A)`, i.e., store value X to the memory location pointed by A. First, for all such store instructions an extra instruction is added of the form `store X',(A')`, where there exists a one-to-one symmetric key mapping between the addresses A and A'. Note that the address A' may be in a random address block, and hence writing a new data to A' might corrupt previously stored data. To prevent this, our system first loads the value from the location A' to a local register and then writes the same value to A'. Thus although the memory block for A' is dirty, its contents remain unaltered. Second, we have to guarantee that these two instructions follow each other in the memory request. At least, we need to guarantee that they need to be close to each other (otherwise, the initial request will be delayed). Therefore, the eviction of one cache block containing A will guarantee the eviction of the cache block containing A'. In such a scenario, every data write request that goes out of the processor to the external memory bus will have the shadow request with address obfuscated by a symmetric key mapping. Third, if the shadow request is not seen for every memory access operation then there could be a possible information leakage attack that has modified the address bus, by inserting a malicious store. Note that such an address mismatch mapping can also take place due to malfunctioning of the processor; however, in either case, we halt the store operation and raise an alarm.

Now let us discuss how such a symmetric mapping to a shadow address can be generated for every address in the program binary. Instructions in a binary use virtual addressing scheme; however, memory transactions always use physical addresses of cache blocks. During virtual to physical address translation, the least significant bits of a virtual address remain unaltered, while the most significant bits are changed to the most significant bits of the corresponding physical address. Hence any address generated by the modification of the least significant bits of a virtual address will reside in the same page in the physical memory. For example, in case of a 4KB page size, the last 12 bits of the any virtual address can be modified to generate an address within the same page in the memory. Likewise, the symmetric key mapping in our scheme changes the last 12 bits of the virtual address to generate an obfuscated address. Note under symmetric mapping with key (K), address A will map to A'= $\boldsymbol{f}$ (A,K) and A' will map to A = $\boldsymbol{f}$ (A',K), where $\boldsymbol{f}$ is a mapping function. A simple way of generating such a mapping is using a xor function; xor operation with a stream of 0s and 1s flips the bits of the original address in location of the 1s and preserves the bits in the location of 0s. One advantage of the x86 ISA is the availability of the xor logical instruction, which makes the address translation an atomic operation. The symmetric mapping in our scheme from address A to A' requires that (i) both go to the same page, (ii) but not to the same block. Hence for our 32-bit system with 4KB page size, the key can only modify middle 5 bits. If a new address is generated by modifying the higher bits of a virtual address then the symmetric mapping may not hold for the corresponding physical addresses. However, if two virtual addresses map within the same page, the symmetric mapping still holds at the physical address level. For this reason, shadow address is generated by performing XOR on the middle 5 bits as indicated in Figure 3a.

4.1. Guaranteeing Predictable Bus Traffic

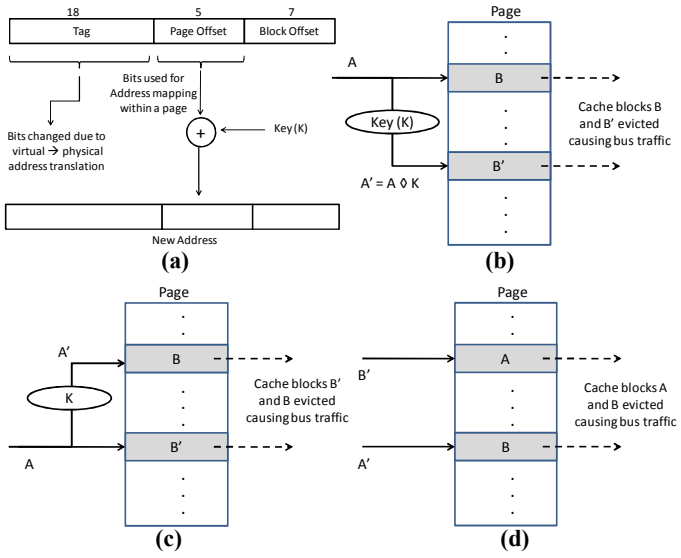

Figure 3. (a) Address mapping/translation assuming 4KB page size and 128B block size in a 32bit system; cache block eviction and traffic generated due to memory access in the order (b) B,B',A,A', (c) B,B',A',A, and (c) A,B,B',A

One of the principal advantages of our scheme is that it generates a detectable memory traffic pattern by guaranteeing the eviction of cache blocks in a sequential order. Codes are instrumented in such a way that a shadow address A' is always evicted once the address A is evicted. To illustrate this idea, let us

assume an address B already loaded in a dirty cache block; according to the scheme this will mean that a corresponding B' exists in the cache block which is symmetrically generated from the original block. In our example, we use the symbol '←→' to imply symmetric mapping, i.e., mapping of instructions to the same cache block within the same page boundary. When another address A tries to store to the same cache block as B (Figure 3b), an extra store to A' is generated such that it accesses the cache block containing B'. This will evict both the blocks containing B and B' and the data traffic will pass through the memory bus. The GC in this case will check whether the mapping between B←→B' holds. If it does not hold, there may be a possibility that a malicious circuit in the chip has affected the original data stream by generating alien data traffic. If it holds, both of these requests will be sent to the memory.

In a second case, Figure 3c, the memory instructions with address A and A' map to cache blocks containing B and B', respectively (here A←→B' and A'←→B). Here the address B' is evicted by A and B is evicted by A' (since an A is always accompanied by an A', and henceforth). The address bus pattern will thus be B' followed by B. The GC can perform its original checking by matching whether the mapping holds.

Things might be more complicated when the address requests are interleaved (Figure 3d). Let us assume the requests come in the following order: A, B, A' and B'. If the mappings A←→B and A'←→B' hold, initially A and B will reside in the cache; after which, A' evicts B and B' evicts A. The bus traffic will thus see A and B coming out which still satisfies the mapping. The GC on finding out that the mapping holds, will allow these operations to go through. Later when A' and B' are also evicted and seen on the bus, these two requests will be forwarded as well. As a result, even if A and A' were not matched together (i.e., the execution was not in the intended order), the execution of the application will continue without causing a delay.

4.2. Binary Instrumentation Framework

To implement the above scheme we have used a static instrumentation technique on x86 binaries. X86 binaries use complex instruction set architecture (CISC) where logical, arithmetic, data transfer, and control flow operations can access memory. Hence for all such memory operations, we have instrumented an address and data mapping technique as described in the previous subsection. The major challenges in designing such an instrumentation framework are due to the steps involved in effective address calculation for every memory access, generating a new address for the same, and generating a data access for the newly generated address. Since all these actions cannot be performed atomically, we insert a set of instructions at specific locations in the basic block. The next task is to find the appropriate location in the basic block to instrument these extra instructions. Specifically, the main bottleneck is to find a "free" register to perform the new address calculation. For this purpose, we use a simple search strategy that checks for dead registers within a basic block. In case a register value is written without being read in the same instruction, that register can be used for instrumentation: the register is used to store the temporary address after mapping (A' for address A). Since the same is rewritten just after the instrumented code the program execution is not affected.

Not all basic blocks contain free registers. In such cases, we consider the control flow to find possibly free registers. Specifically, if the current basic block falls through to the next one, we can find a dead register in the following block and use it in the current block for address calculation (assuming there are no uses of the register in the following block). In addition, if the current basic block ends in a conditional branch operation, free register in the fall through basic block may not be used. In this case, we search for the necessary remote basic blocks and check whether a register is free in all such subsequent basic blocks. If successful, we use this register in the current basic block for instrumentation. If all these three cases fail, we generate an additional global memory, as a temporary scratchpad for the address conversion. Note that we have to perform two load and store operations to this location. Hence, the main disadvantage of this option is the performance overhead. A memory operation is much more expensive than a logical register operation. Figure 4 provides a pictorial description of the code instrumenting strategies we adopt for our scheme. Note that this method is similar to the live variable analysis and register allocation [6] performed by state of the art compilers. However, since we are using a binary instrumentation framework to incorporate security features, we are adding this extra overhead.

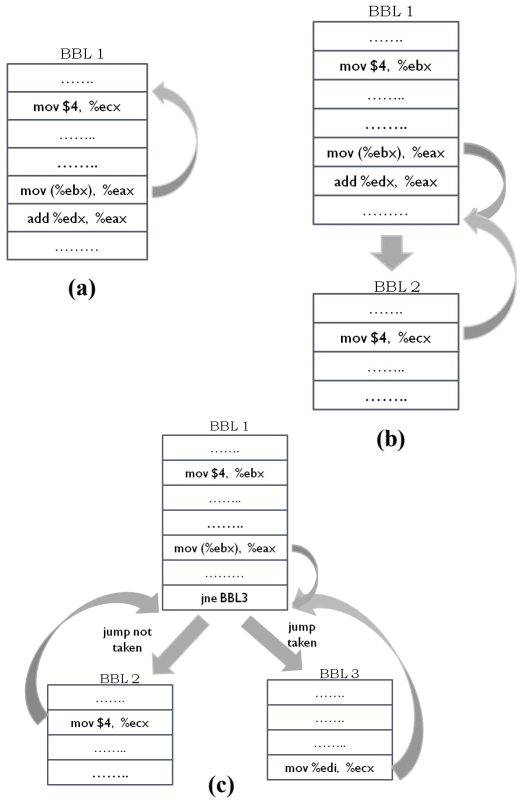

Figure 4. Instrumentation using a free register from (a) same BBL, (b) fall through BBL, and (c) BBLs from possible control flow

For our framework to work, the system must satisfy the following criteria. First, the L2 cache write policy has to be write through, i.e., the updated information in the L2 cache is written to both the block in the cache and to the corresponding block in the memory. If such assumption does not hold true, then there can be a possible interleaving of data during eviction. Secondly, the cache eviction policy is assumed to be deterministic. In other words, our scheme will work for policies like Least Recently Used (LRU) block eviction and not random eviction.

4.3. Guardian Core (GC) Functionality

The GC is a reprogrammable unit consisting of a local storage buffer and simple control logic. The GC lies in between the

memory and the processor, and can be thought of a secondary core (soft core), or a component of the memory controller that monitors the bus activity generated by the processor. The local buffer is a content-addressable memory (CAM) that stores address and data blocks from the bus. The maximum size of this buffer is limited by the total number of blocks in the cache. However, for most applications, we see that a smaller buffer size would suffice. In case several applications share the same cache, the size of the buffer can be determined by the total number of blocks used by the applications. Besides, the GC also possesses a simple control unit, which is capable of performing logical operations. This unit is used during address checking operation when a given address value is XOR'd with the user provided key checked against the addresses in the buffer.

For a load operation, the GC immediately sends the request to the memory. At the same time, it checks whether the requested address is present in its local buffer. If it finds this data, it is forwarded to the processor and the new data arriving from the memory is ignored. During a store operation, the GC checks the store address with the address entries in its local buffer. In case the GC sees an address that maps to a stored address using the key mapping function, it evicts the entry since a matching pair has been found. In such a case, both of the write operations are carried out. If the address on the bus doesn't match with any of the buffer entries, it stores the address and data as a new buffer entry. The scheme ensures that every store to address A on memory bus will have the matching A' evicted from the cache. The GC thus waits for the equivalent A'. Section 5 provides a listing on the number of instructions between the original and the instrumented instruction.

In case the GC doesn't find a match for any buffered address at the end of program execution, this will mean that a store instruction within the application is modified into a new store instruction with a different address for which the mapping function does not hold. In other words, the Trojan hardware has inserted a new store instruction in the stream. Under such circumstances the GC raises an alarm that a malicious bus activity has taken place.

The latency of a load is unaltered due to the GC. In fact, it may speed up the data access by providing dirty data from its local buffer. On the other hand, during a store operation, there is an additional overhead due to GC's checking of addresses. However, store instructions are generally not "urgent". In other words, if the processor is writing a dirty block, the store instructions can be delayed without affecting the rest of the execution in the core. Note that we assume that the GC does not collude with the main processor to leak information. Also, since the GC sits outside the main chip, which may have a malicious hardware inside, it is out of the reach of the attacker. Moreover, making the functionality of GC to be reprogrammable makes it less susceptible to attacks.

5. Experimental Results

We developed a full system prototype of an x86 static binary instrumentation framework on a dual core 2.6 GHz AMD Opteron processor with 1MB L2 cache. We have currently implemented our framework on Suse-10 Linux environment with *gcc* compiler version 4.1. Our prototype system takes in a *gcc* compiled binary, disassembles it into assembly code and instruments the same, and recompiles to form the modified binary. To examine the effectiveness of our approach we chose several x86 elf format binaries from three different benchmark suites. The first one is MiBench, a free, commercially representative embedded benchmark suite which encompasses a wide variety of applications from sorting to complex hashing algorithms [7]. The second benchmarking suite used is MineBench which is a collection of data mining applications extensively used to solve commercial decision making problems [8]. The final benchmark suite used is Stanford parallel applications for shared memory (Splash2) [9], which consists of a set of data parallel scientific applications and kernels.

Table 1 describes the instrumentation details for the studied applications. Most of applications have a considerably high fraction of memory instructions. The ratio of the number of memory instructions to the total number of instructions is usually more than 30% for the selected applications. In general, we observe that most memory instructions can be instrumented by finding a free register within the same basic block: 74.0% of all the memory instructions are instrumented with this method. On the other hand, we see that the control flow analysis mechanisms are not highly successful: only 2.8% of the encountered memory instructions are instrumented with these techniques. The remaining memory instructions (23.3%) are instrumented using the additional accesses to the global memory.

The instrumentation stage determines the length of the queue in the GC. Table 2 gives the depth of the maximum queue length necessary for each application in order that the GC receives the matching memory instruction. It is clear that with small buffer size of 7 to 51, the GC can perform the checking operation.

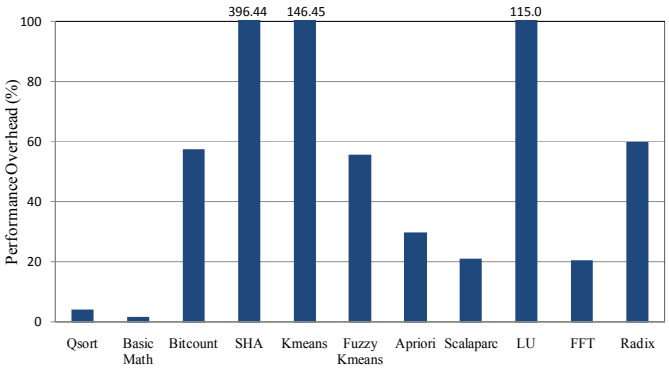

Figure 5. Performance overhead for different Minebench, MiBench, and Splash2 applications

The results for the performance overhead are summarized in Figure 5. Overall, we see widely varying levels of overheads for the studied applications. For some applications, the overhead is less than 5% (Qsort and BasicMath). On the other hand, the overhead can be as high as 396.4% (for the SHA algorithm). In general, we see that the compute-bound applications present a significantly lower overhead when compared to data-intensive applications.

One of the reasons why the performance overhead varies considerably is because of the dataset sizes of the applications. Hashing algorithms (SHA) or clustering algorithms (Kmeans) are typically memory intensive, and use a large input dataset. Hence even a small increment in the data size due to addition of global memory used in instrumentation will cause the data cache to be flushed at regular intervals. This increases the overhead due to instrumentation. On the other hand, applications like Qsort and BasicMath use considerably small datasets; and thus show lower overheads. Note that in the case of a multi-tasking environment, the instrumented application binary would affect other applications as the cache would be utilized more. However, in this study we do not consider such cases for the sake of simplicity.

Table 1. Memory instruction profiling information

Benchmark	Application	Total # of Instructions	Total # of Memory Accesses	# of Free Register from same BBL	# of Free Register from fall through BBL	# of Free Register from two possible BBLs	# of References to global memory
MiBench	Qsort	152	60	53	1	1	5
	Basic Math	662	370	87	0	0	283
	Bitcount	465	108	63	0	3	42
	SHA	530	226	205	9	2	10
MineBench	Kmeans	2855	1663	1300	45	24	294
	Fuzzy Kmeans	2855	1663	1300	45	24	294
	Apriori	3467	1574	1055	19	3	497
	Scalaparc	1946	1001	787	12	15	187
Splash2	LU	1855	883	663	5	19	196
	FFT	2138	938	728	4	13	210
	Radix	1708	678	541	11	6	120

Table 2. Maximum necessary queue length in the guardian core (GC)

Max. Queue Length	Application										
	Qsort	BasicMath	Bitcount	SHA	Kmeans	Fuzzy Kmeans	Apriori	Scalparc	LU	FFT	Radix
	7	51	11	9	17	17	16	37	16	32	23

Another reason for high overheads for some applications can be due to their dynamic nature of execution. If some instrumented parts of a binary that uses global memory are executed frequently due to spatial locality of execution, there is a higher possibility that the frequent access of global memory will slow down the execution. Since the instrumentation is performed statically, this situation is beyond our control.

6. Related Work

The bulk of techniques in computer security lie in software approaches that include programming languages, obfuscation, operating system security, and intrusion detection. However, recently, researchers have shown interest to the malicious hardware problem [2, 10, 11]. King et al. demonstrate the 'Illinois Malicious Processor' where they bypass the secure login of the root by using shadow registers and malicious memory controller [10, 12]. Simpler forms of attacks have been shown by Agrawal et al. [2] where they implement the Trojan circuit using a 16-bit counter which counts up to a certain threshold value and then disables parts of the circuit. Besides there are several web articles which talk about the issue of having a backdoor switch in hardware [1]. Roy et al. talks about hardware piracy and IP loss [13]; but their attack model is orthogonal to ours since we attack. To the best of our knowledge there has been no prior work in the area of information leakage detection due to malicious hardware and thus we pose a new problem to the systems community.

7. Conclusions

Processors manufactured in un-trusted foundries can cause a security breach as the processor core itself may be modified. One such security vulnerability can be exploited when a malicious Trojan circuit inside a processor leaks confidential information, by thwarting the usual instruction stream of an application. In this paper we propose a new framework for detecting address bus tampering using malicious hardware. It employs static binary instrumentation technique by which every memory load store is mapped to a corresponding shadow operation which can be easily detected by an external guardian execution core. We demonstrate that the performance of such a scheme can range from 1.7% to 396.4% (average 82.5%) depending on the nature of the application. Lastly we infer that binary instrumentation techniques can be used as useful and flexible mechanisms for preventing hardware Trojan attacks.

References

[1] S. Adee, "The Hunt for the Kill Switch " Available at http://spectrum.ieee.org/may08/6171; IEEE Spectrum Online, May 2008

[2] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan Detection using IC Fingerprinting " in *IEEE Symposium on Security and Privacy (SP)*, Berkeley, CA, 2007, pp. 296-310.

[3] "DS5002FP secure microprocessor " 1998.

[4] A. Huang, *Hacking the Xbox: An Introduction to Reverse Engineering*: No Starch Press, 2002.

[5] B. Yang, K. Wu, and R. Karri, "Scan based side channel attack on dedicated hardware implementations of Data Encryption Standard," in *International Test Conference (ITC), 2004.* .

[6] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers: Principles, Techniques, and Tools ", 2006 edition ed.

[7] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *4th Annual Workshop on Workload Characterization*, Austin, TX, 2001.

[8] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, J. Pisharath, G. Memik, and A. Choudhary, "MineBench: A Benchmark Suite for Data Mining Workloads", in *IISWC, 2006*.

[9] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *22nd International Symposium on Computer Architecture*, Margherita Ligure, Italy, pp. 24-36.

[10] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware," in *First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*.

[11] R. Simha, B. Narahari, J. Zambreno, and A. Choudhary, "Secure Execution with Components from Untrusted Foundries," in *Advanced Networking and Communications Hardware Workshop*, 2006.

[12] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious processors," in *Wild and Crazy Ideas Session VI (ASPLOS XIII)*, 2008.

[13] J. A. Roy, F. Koushanfar, and I. L. Markov, "EPIC: Ending Piracy of Integrated Circuits," in *Design Automation and Test in Europe*, 2008.