Appears in the First Workshop on Compiler and Architectural Techniques for Application Reliability and Security (CATARS)
In Conjunction with the 2008 International Conference on Dependable Systems and Networks (DSN 2008)
Anchorage, Alaska, June 2008

IOTA: Detecting Erroneous I/O Behavior via I/O Transaction Auditing

Albert Meixner and Daniel J. Sorin
Duke University
albert@cs.duke.edu, sorin@ee.duke.edu

Abstract

The correctness of the I/O system—and thus the correctness of the computer—can be compromised by hardware faults, driver bugs, and security breaches in downloaded device drivers. To detect erroneous I/O behavior, we have developed I/O Transaction Auditing (IOTA), which checks the high-level behavior of I/O transactions. In an IOTA-protected system, the operating system creates a signature of the I/O transactions it expects to occur and every I/O device computes a signature of the transactions it actually performs. By comparing these signatures, IOTA can discover erroneous behavior in both the hardware and the software responsible for performing I/O operations. Our experimental results show that IOTA reliably detects a wide range of erroneous behavior at little cost in performance, I/O bandwidth, or hardware.

1. Introduction

As computer system dependability has become increasingly important, there has been a significant amount of research in checking the system's behavior. Recent research has primarily focused on checking the behavior of cores [2, 8] and memory systems [9], while I/O systems have received comparatively little attention.

The I/O system includes both hardware and software, and it comprises a crucial part of a computer system. I/O hardware consists of the actual devices (disks, network cards) and the interconnect (e.g., PCI, AGP, ISA, HyperTransport) that enables communication between the devices, processor, and memory. I/O software includes the device drivers that provide the interface between the OS and the devices.

The I/O system can behave erroneously due to three causes. First, like cores and memory systems, I/O hardware can suffer from physical faults, both transient and permanent. Second, the I/O system is susceptible to device driver bugs. Third, I/O may behave erroneously due to security breaches, particularly in downloaded device drivers.

Our goal is to detect erroneous I/O behavior. Once detected, the system can take action to diagnose and remedy the situation. If a permanent device fault is diagnosed, the system could suggest that the user or system administrator replace this device. If a driver bug is diagnosed, the bug can be reported and the driver can be restarted (which is often sufficient). If a security breach is diagnosed, the system can shutdown the driver under suspicion and alert the user. In this work, we focus on error detection, rather than diagnosis and recovery.

Our error detection mechanism, *I/O Transaction Auditing (IOTA)*, uses end-to-end checking [13] of I/O *transactions*. We define a *transaction* as a high-level, semantically atomic I/O operation, such as sending an Ethernet frame. A transaction represents the basic I/O operation for higher level OS services such as the file system or network stack. A transaction typically consists of many low-level I/O operations, such as reading or writing memory-mapped device registers. For example, sending an Ethernet frame is a transaction that consists of multiple I/O accesses from the driver to initiate the transaction at the device and many memory accesses by the device to obtain the frame payload. Transactions are high-level operations that are independent of low-level implementation details; for example, a disk transaction is independent of the format of the file system.

The basic idea of IOTA is to audit I/O transactions at multiple points in the system (OS, buses, devices) to ensure that they are propagated correctly through the I/O software and hardware stack. In an IOTA-protected system, the OS creates a signature of the I/O transactions it expects to occur. IOTA's auditing hardware—which could be located at an I/O device (e.g., SCSI device) or I/O controller (e.g., SCSI controller for multiple SCSI devices)—computes a signature of the transactions the device actually performs. By comparing these signatures, IOTA can discover many types of erroneous behavior between the OS and the auditors. Moreover, IOTA can detect many errors not detectable by simpler mechanisms such as error detecting codes (signatures over only the transmitted data) or higher-level mechanisms (e.g., timeouts on dropped packets). For example, IOTA can detect if a malicious driver tries to send data to an attacker.

IOTA's primary contribution is an end-to-end error detection scheme for I/O. It detects many faults, driver bugs, and security breaches. However, it has some limitations, including a device model that is not entirely

general and excludes some common I/O devices, such as graphics accelerators.

In the rest of this paper, we describe and evaluate IOTA. First, in Section 2, we describe our error model. Then, we present IOTA in Section 3, and we evaluate it in Section 4. We discuss related work and compare it to IOTA in Section 5, before concluding in Section 6.

2. Error Model

We consider errors due to three causes: physical faults, driver bugs, and driver security breaches. These errors all occur between the OS and IOTA's hardware auditors, which enables detection of many of them.

2.1 Physical Faults

Our fault model is a single stuck-at fault that is either transient or permanent. Due to faults, a variety of errors can occur, and we provide examples:

- Incorrect Actions: A device sends a network packet to the wrong destination.
- Out-of-order Actions: The OS specifies that the disk controller should complete the DMA before sending an interrupt, but the interrupt occurs first.[1]
- Missing/Duplicated Actions: The network device simply does not send a packet.
- Wrong Status: A device's status is incorrect. This error is subsumed by the previously listed errors.
- Deadlock/Hanging: A device does nothing.

These erroneous behaviors are not equally likely, even though we assume the underlying faults are equally likely to occur in any hardware component. Although some of these errors are easily detectable with other mechanisms (e.g., an error in writing data to a disk is detectable with EDC), others errors are more challenging (e.g., an error that causes a write to a disk to not be performed at all).

2.2 Driver Bugs

Device driver bugs are common [3] and can be expected to remain a problem in the future for a variety of reasons. Most notably, there exists a vast number of drivers, each of which has to manage a complicated interface between the OS and the device. The device driver has to split a semantically atomic operation, such as writing a block to disk, into a sequence of requests to the device. While doing this, it must also obey device idiosyncrasies (such as timing constraints) and handling asynchronous events (such as interrupts). This requires sophisticated synchronization and complex logic for infrequent special cases (error handling), which makes thorough testing difficult. A recent report from Coverity showed that 53% of detected bugs in the Linux kernel were in device drivers [4].

When a driver bug is excited, it manifests itself as an erroneous behavior. Because the space of possible design bugs is unbounded, we cannot construct a complete error model. One can always imagine a more diabolical bug that does not fit into a given error model.

2.3 Security Breaches

Drivers are a natural weak point in a trusted OS environment. Although their access to memory can be restricted in the same way as for regular applications, drivers by necessity have access to the devices they are controlling and to user data transferred between the device and the system. This position can be exploited to maliciously leak sensitive data. Unlike other trusted components, such as the OS kernel, drivers are provided by many sources and are frequently updated. Thus, the often low code quality that manifests itself in numerous bugs also makes drivers suspects for possible intrusions. A widely publicized recent breach in an Apple MacBook was possible when a downloaded third-party driver was used for its wireless Ethernet card [14]. This breach was not preventable by use of a personal firewall.

The damage that a driver can cause depends on the device. Drivers for communication devices like network cards can analyze data transferred over the Internet by the user, such as e-mail, and communicate the collected information to an outside party. Encryption alone is insufficient to counter malicious drivers, because the driver has full control over incoming and outgoing data and can thus execute man-in-the-middle attacks. Mass storage drivers (SCSI, IDE, etc.) do not have the same ability to communicate data to an outside party, but they can help an attacker circumvent the operating system file permissions in order to access confidential data.

3. I/O Transaction Auditing (IOTA)

In this section, we present IOTA's system model, give an overview of how IOTA fits into the remainder of the system, discuss our particular implementation, and explain IOTA's costs and limitations.

3.1 System Model and Key Insight

IOTA views each I/O device as a complex functional unit that performs a series of self-contained input and output transactions. A device performs a transaction either as a result of commands issued by the processor or due to outside events, e.g., incoming messages. Once a transaction completes, the device notifies the CPU via an interrupt or polling and makes the results accessible to the processor.

IOTA leverages the observation that, at a high level, many I/O devices are quite simple and merely convert data from one representation into another without add-

1. IOTA detects out-of-order device actions, but it does not prohibit re-ordering of transactions unless required by the OS.

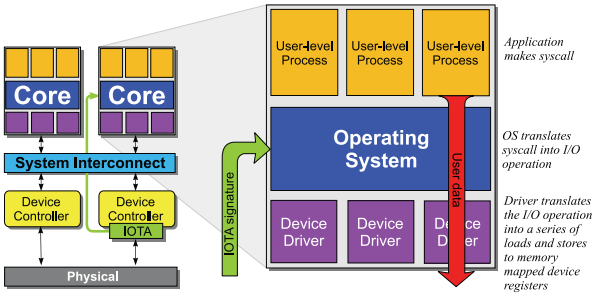

Figure 1. IOTA Hardware & Software schematic

ing information. An Ethernet controller, for example, simply transforms a frame stored in memory into a format that is suitable for transmission on a the shared Ethernet bus. The complexity arises due to two reasons. First, there are inefficiencies in the communication interfaces between devices, processors, and memories. Second, there is the necessity of serializing a number of asynchronous requests to a shared medium.

IOTA's auditing hardware does not need to deal with serialization, because it simply observes the already serialized device output. It also avoids communication inefficiencies by using signatures to represent transactions, which allows a transaction to be represented with a single data word that can be atomically transmitted to the processor.

3.2 IOTA Design and Operation

An overview of IOTA's integration into both the hardware and software stack is illustrated in Figure 1. For each I/O transaction, the OS computes a signature of the operation it expects the device to perform (outgoing transaction) or that it observed the device perform (incoming transaction). The auditor at the device computes a signature of the transactions that the device completes. The OS can query the auditor to obtain the device's signature and compare it to its own signature for a given transaction. If the signatures differ, an error has been detected. The OS requires a small amount of storage for holding signatures of outstanding transactions. The IOTA signatures are analogous to a checksum (e.g., CRC) over a transaction, including its control, rather than the typical checksum over data.

As described, the erroneous I/O would still occur, but it would be detected immediately afterwards. Immediate detection prevents any further erroneous I/O until the system regains confidence in the device and driver (e.g., through testing). If a system cannot afford even a single erroneous I/O transaction, then we could slightly modify IOTA to serve as a gate that only allows I/O to be performed once the OS sends a matching signature.

3.3 Security Features

When discussing IOTA's security benefits, we assume that the operating system restricts the privileges

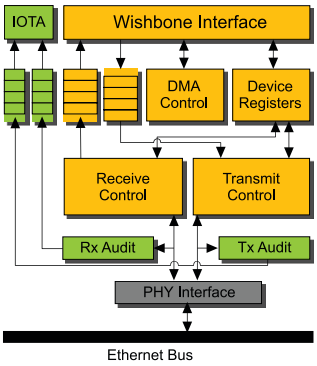

Figure 2. IOTA implementation for Ethernet controller

of drivers just as it would for other applications. Device drivers with full kernel access rights have many opportunities for causing harm that are simpler than the rather sophisticated exploits IOTA is trying to prevent. Even in such a system, IOTA as described thus far could be easily circumvented by two different techniques.

Circumvention #1. If the interface for reading signatures from the auditor is controlled by the driver, like the rest of the device's interface, an untrustworthy driver could intercept requests from the OS, compute the expected signatures, and provide these seemingly correct signatures to the OS. To detect these breaches, we add a direct interface between the OS and the device that allows the OS to directly read the device's IOTA signature from a memory-mapped device register. The driver is kept from accessing this interface using standard page protection mechanisms. This interface is fundamentally necessary to prevent an untrustworthy driver from providing a facade of correct behavior.

Circumvention #2. The driver may try to avoid detection by constructing a transaction that is different from the transaction initiated by the OS, yet has the same signature. IOTA can thwart such an attempt by using a key-dependent algorithm to compute message signatures. The hardware auditor and the OS exchange the keys used to generate signatures through the direct interface for exchanging the signatures that was described earlier.

3.4 Implementation Details

Our sample implementation of IOTA, illustrated in Figure 2, is based on an open-source Ethernet controller obtained from OpenCores.org [7]. The Ethernet controller is connected to an OpenRISC 1200 [7] CPU and memory through a wishbone interface. The Ethernet chip is configured and controlled via memory-mapped I/O, and it accesses frame data through direct memory accesses. We have implemented the hardware support for checking the Ethernet device in Verilog. The I/O auditor modules are located at the interface between the Ethernet controller and the physical interface module

that transforms the binary controller output into a format suitable for transmission on an Ethernet cable.

The implementation used in our experiments computes a signature using a 16-bit cyclic redundancy check (CRC) that summarizes the type of transmission (send/receive), frame headers, frame payload, and relevant configuration parameters set at the time of the transmission. The signature could easily be widened to 32-bits or more, but we found the error detection capability of a 16-bit signature sufficient for our purposes.

We leave the application of IOTA to other devices for future work, but the fundamental structure of all implementation will be similar across different devices.

3.5 Costs

The increase in dependability provided by IOTA is not free, as IOTA incurs three types of costs.

CPU time. The OS must compute its signature for each I/O transaction. We empirically determine the impact of this cost on system performance in Section 4.3.

I/O bandwidth. I/O bandwidth is used when the OS queries a device for its signature and when the device responds with it. However, signatures are small and comprise a miniscule fraction of I/O bandwidth.

Hardware to compute signatures at devices. We add a small amount of hardware to the devices so they can compute and maintain their signatures. The two auditors comprise about 5% of the number of logic cells occupied by the synthesized, SRAM-less Ethernet controller. IOTA's overhead is less if the controller's on-chip SRAM is taken into account.

3.6 Limitations

Although IOTA was designed as a generic mechanism for detecting a wide range of incorrect I/O behavior, it has several limitations. First, IOTA's system model is not entirely general and there are some classes of devices not suitable for IOTA. In particular, devices that maintain large amounts of state and do complex computation on their own—such as graphics (e.g., NVIDIA GeForce [10]), TCP, and physics accelerators (e.g., ParallAX [15])—are difficult to fit into the IOTA model. Our primary targets in developing IOTA were simple mass storage and communication devices, like disk controllers and network cards. These types of controllers are present in all desktop and server machines, and they exist in some form in most special-purpose and embedded systems. Furthermore, failures in these devices are likely to corrupt user data or leak sensitive information to the outside world.

Even for devices that are contained in IOTA's system model, IOTA cannot detect all possible errors in I/O behavior, due to three primary reasons:

- IOTA uses fixed-size (16-bit) signatures with a non-zero probability of aliasing.
- The space of possible errors caused by security breaches and driver bugs is unbounded.
- In some I/O interface architectures, a device can stop the system by not releasing the I/O bus. In this case, the OS cannot successfully query the device's auditing unit and, thus, IOTA provides no benefit.

Table 1. Error Injection Results

%age	Observed Behavior
42.3	Completed successfully, no observed error
18.7	Transmitted data contained errors
15.1	Received data contained errors
9.7	Controller didn't recognize incoming transmission
7.3	Test completed successfully, but faulty IOTA hardware reported an error
2.2	Transmitted and received data had errors
2.2	Transmitted frame was incomplete
1.3	Controller did not transmit frame, but reported completion of transmission
0.9	Controller stalled Wishbone Bus
0.3	Controller stalled Ethernet bus by sending garbage

4. Evaluation

The primary goal of this evaluation is to demonstrate that IOTA detects erroneous behavior in the I/O system. We also evaluate the overheads that IOTA introduces, in terms of performance and I/O bandwidth.

4.1 Detecting Errors due to Faults

For our error injection experiments, we simulate the Ethernet chip along with the I/O auditing modules at the gate level. Faults were injected by corrupting the behavior of a randomly selected 1,000 out of about 20,000 total gates. The OR1200 CPU is simulated at the level of functional Verilog. The experiments are run in a Verilog testing harness that sends Ethernet frames to the simulated system and monitors outgoing frames on the Ethernet bus. The CPU executes a test routine that emulates the behavior of a system forwarding a frame.

The experimental results are summarized in Table 1. Among errors that affect the test outcome, data errors are by far the most frequent. This is unsurprising, because most of the device's logic is used for data buffering, data transfer, and arbitration. Another frequent error scenario is the device not executing an operation, by either completely or partially failing to send a frame or by missing an incoming frame. Fortunately, the worst kinds of errors, which cause the controller to stall either the Ethernet bus or the system (i.e., Wishbone) bus are very infrequent. IOTA successfully detected all true error cases, except those that prevented the CPU from

querying IOTA by stalling the system bus. IOTA incurred a number of false positives when an error was injected into the auditing hardware itself; however, the percentage of errors causing false positives is overemphasized in this test by not injecting errors into the buffer descriptor RAM, which makes up a significant portion of the Ethernet controller.

4.2 Detecting Errors due to Device Driver Bugs and Security Breaches

Both device driver bugs and security holes are design flaws and thus more difficult to quantify than permanent and transient hardware faults. Trivial driver bugs are likely to cause page access violations or faulty behavior at the driver/OS interface, which can be found using software techniques. Subtle logic bugs and synchronization problems, which can cause incorrect command sequences to be sent to the device, can easily stay hidden from the OS, but are easily detectable using IOTA. These types of failures are difficult to produce in an automated, realistic manner. Security breaches are even more difficult to model, because they require both a software weakness and a malicious attacker willing to exploit it. Therefore we do not currently have quantitative injection results for these types of errors.

Security breaches are likely to try to leak information and will thus appear as data errors to IOTA. Hardware error results show that IOTA can reliably detect data errors. Driver bugs can exhibit more varied behavior, but the overall error categories are similar to hardware faults, although they will be distributed differently. The fact that IOTA reliably detected all kinds of errors in the hardware test suggests that it will perform well in detecting driver bugs, too. Nevertheless, future work is required to confirm IOTA's effectiveness.

4.3 Performance Overhead

IOTA could potentially degrade performance due to the time required for the OS to update its signature. To determine this performance overhead, we implemented the software side of IOTA as a load-able kernel module for Linux 2.6.16. The module computes the signature for every incoming and outgoing Ethernet frame. To measure performance, we ran netperf [6] on a Dell PowerEdge server with a 1.4Ghz Intel Pentium 3, 1GB RAM, and an Intel 82544EI Gigabit Ethernet Controller using Linux's e1000 driver module.

Table 2 summarizes the results of measuring a single TCP connection at maximum speed. We compare the base configuration (without IOTA) to both a basic IOTA implementation that computes the signatures entirely in software and a hypothetical implementation that uses a specialized checksum (CRC) update instruction with a throughput of 1 word per cycle. We consider hardware acceleration for CRC to be a realistic possibility; Intel recently introduced a CRC32 instruction into its IA32 instruction set [12], and instructions to accelerate CRC computation have been added to specialized processors, such as the Intel IXP network processor [1].

Table 2. IOTA Software Performance

	Config	Throughput MBit/s	CPU load	CPU service time (µs/KB)
Send	Base	293	20.8%	5.8
	IOTA	272	43.6%	13.1
	IOTA/Accel	293	26.6%	7.4
Receive	Base	713	90.0%	10.3
	IOTA	423	73.7%	14.2
	IOTA/Accel	652	90.0%	11.3

The results show that, when not provided hardware acceleration, IOTA can significantly degrade performance in worst-case scenarios, but will have little impact in most cases. Signature computation requires about 4-7 µs of CPU time per KB transmitted or received. Consequently, throughput would become CPU bound at about 600 Mb/s, whereas the base system can receive data at up to 800Mb/s before becoming CPU limited and send at the full line speed of 1Gb/s. Thus a pure software IOTA implementation would not be suitable for routers or high-performance server that can process requests at near full line speed. However, static web servers only reach a fraction of theoretical throughput (~100Mb/s) and the throughput of dynamic web servers is closer to 10Mb/s [11]. The transmission rates of typical desktop and laptop computers are significantly lower than what Gigabit Ethernet provides. At these rates, CPU utilization by the network stack, even with software IOTA, would be below 10%.

Hardware acceleration reduces the overhead of signature computation to about 1 µs/KB. This makes IOTA feasible even for systems that require very high network throughput and for handheld and embedded systems with less processing power.

4.4 I/O Bandwidth Overhead

IOTA uses some additional I/O bandwidth to communicate signatures between the OS and the devices. The OpenCores Ethernet controller used in our Verilog implementation, when configured to use interrupts to signal operation completion, requires three writes, one read, plus frame payload to perform a send and three writes, two reads, plus payload to receive a frame. IOTA adds one read access to the necessary sequence of accesses. Assuming the smallest Ethernet frame size of 64 bytes, IOTA's bandwidth overhead is 5% for sending and 4.7% for receiving. For larger frame sizes, IOTA's bandwidth overhead drops to well below 1%.

5. Related Work

The most closely related work is Saltzer et al.'s [13] classic paper on end-to-end checking in system design. They observe that end-to-end checking is a powerful technique for comprehensive detection of errors, and it is preferable to cobbling together multiple localized error detectors. Our work builds on this paper by developing an end-to-end checker for the I/O system.

One extremely common mechanism for detecting I/O errors is the use of error detecting codes (EDC) and error correcting codes (ECC) on links, messages, and storage. IOTA is more comprehensive than EDC/ECC, because EDC/ECC only detects errors in a given link or piece of storage; EDC/ECC cannot detect when, for example, a correct piece of data is sent to the wrong device or written to an incorrect location.

A more sophisticated and flexible framework for detecting I/O errors is I/O shepherding [5]. I/O shepherding, which is geared towards file system reliability, provides the file system developer with a mechanism for detecting and handling errors. The developer can construct various policies for the shepherd. The shepherd sits between the file system and the disks, unlike IOTA which is more of an end-to-end approach; IOTA can thus detect additional types of erroneous behavior.

6. Conclusions

IOTA is a first attempt at providing a general, low-cost, and end-to-end mechanism to ensure that the actions of an I/O device match the original intent of the OS. IOTA's end-to-end approach enables it to detect multiple dependability problems (hardware faults, driver bugs, and security breaches) in a number of both hardware (device, I/O controller, I/O bus) and software (driver) components. IOTA does not make existing protection techniques obsolete, but instead efficiently closes holes left by these mechanisms. For example, IOTA will be of little use to protect against driver security problems, if the drivers execute with full kernel privileges and have full access to the entire space. However, in a system that already uses sandboxed drivers, IOTA can close the remaining security hole, which exists because the drivers have full and unchecked control of their respective devices. This hole cannot be closed by software techniques alone.

IOTA is a light-weight technique; it requires small amounts of additional hardware and only minor changes to existing operating systems. A side effect of this light-weight design is a restricted system model, which limits the range of devices that can be fitted with IOTA. However, we do believe that IOTA is flexible enough to fit two of the most common and important classes of I/O devices: mass storage and network communication. IOTA causes a non-trivial increase in CPU utilization, but we have shown that this problem is limited to high-performance networking and can be alleviated using minor extensions to the CPU instruction set. Nevertheless, CPU utilization is a problem that warrants further investigation, especially when trying to apply IOTA to high-end storage and networking solutions.

References

[1] M. Adiletta et al. The Next Generation of the Intel IXP Network Processors. *Intel Technology Journal*, 6(3):6–18, Aug. 2002.

[2] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[3] A. Chou et al. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 73–88, Dec. 2001.

[4] Coverity. Analysis of the Linux Kernel. Case Study, Dec. 2004.

[5] H. S. Gunawi et al. Improving File System Reliability with I/O Shepherding. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Oct. 2007.

[6] R. Jones. Netperf: a Network Performance Benchmark. http://www.netperf.org/netperf/NetperfPage.html, Feb. 1996.

[7] D. Lampret. OpenRISC 1200 IP Core Specification, Rev. 0.7. http://www.opencores.org, Sept. 2001.

[8] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2007.

[9] A. Meixner and D. J. Sorin. Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 73–82, June 2006.

[10] J. Montrym and H. Moreton. The GeForce 6800. *IEEE Micro*, 25:41–51, March/April 2005.

[11] J. P. Patwardhan, A. R. Lebeck, and D. J. Sorin. Communication Breakdown: Analyzing CPU Usage in Commercial Web Workloads. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 12–19, Mar. 2004.

[12] R. M. Ramanathan. Extending the World's Most Popular Processor Architecture. Intel White Paper, 2006.

[13] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in Systems Design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[14] J. C. Welch. Has Apple Lost Its Security Shine? *EE Times*, September 18 2006.

[15] T. Y. Yeh, P. Faloutsos, and S. J. Patel. ParallAX: An Architecture for Real-Time Physics. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 232–243, June 2007.