The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems*

Olin Sibert †
sibert@oxford.com
Oxford Systems Inc.
30 Ingleside Road
Lexington, MA 02173-2522

Phillip A. Porras
porras@aero.org
The Aerospace Corporation
2350 East El Segundo Blvd
El Segundo, CA 90245-4691

Robert Lindell
lindell@aero.org
The Aerospace Corporation
2350 East El Segundo Blvd
El Segundo, CA 90245-4691

Abstract

An in-depth analysis of the 80x86 processor families identifies architectural properties that may have unexpected, and undesirable, results in secure computer systems. In addition, reported implementation errors in some processor versions render them undesirable for secure systems because of potential security and reliability problems. In this paper, we discuss the imbalance in scrutiny for hardware protection mechanisms relative to software, and why this imbalance is increasingly difficult to justify as hardware complexity increases. We illustrate this difficulty with examples of architectural subtleties and reported implementation errors.

1 Introduction

This paper reports on the progress of an in-depth analysis project covering the 80x86[1] families of processors. This includes the Intel 386, Intel 486, and Intel Pentium[2] processors, as well as their compatible brethren developed by AMD, Cyrix, NexGen, etc. The analysis is based entirely on publicly available materials and tests developed for the project.

This analysis is being performed under the auspices of the National Security Agency's Trusted Product Evaluation Program (TPEP). Because the preponderance of products in or being considered for TPEP evaluation use 80x86 processors (particularly high-assurance products), the intent is to provide a common base of analysis that can be used by evaluation teams for different products incorporating these processors.

Traditionally, computer security evaluations have devoted little attention to hardware.[3] Section 2 explores this approach of implicitly trusting a system's hardware layer with minimal analysis, and why the increasing complexity of modern microprocessors makes this practice untenable.

The initial stage of the analysis focused on aspects of the architecture that may present pitfalls for secure system designers. Although we found no gross security flaws "required by the specification," we identified several features that, if not properly managed, introduce previously unreported covert channels and other subtle problems. These results were surprising; we did not expect well-defined architectural features to cause undesirable security behavior. In retrospect, however, the very complexity of the architecture suggests that it was bound to include some unexpected feature interactions. These problems are discussed in section 3.

In addition, we collected numerous reports of implementation errors claimed to exist in some processor versions. These are summarized in section 4. Although we were aware[4] of a few widely-publicized computational errors in some versions (e.g., incorrect results for 32-bit multiplications), we were surprised to learn of so many distinct and varied reported implementation errors. Because some of these errors could introduce an exploitable security flaw into a secure sys-

*The work reported in this paper was performed by The Aerospace Corporation under a contract to the National Security Agency as part of the Trusted Product Evaluation Program (contract F04701-93-C-0094).

†Oxford Systems participated in this study under contract to The Aerospace Corporation.

[1] The "80x86" designation in this paper means, roughly, "80386 or better." The 8086 family is not included because it has no security mechanisms. Although the 80286 includes some of the same security mechanisms as later processors, it is not included because its architecture is significantly different and it is not being used for new product development.

[2] Intel 386, Intel 486, and Pentium are trademarks of Intel Corporation. All other trademarks are the property of their respective owners.

[3] The term "hardware" in this paper refers both to hardware and firmware, as they are inseparable for modern microprocessors.

[4] The recent publicity surrounding the floating point divide problem in Pentium processors came after the work reported by this paper, but is certainly now the foremost example of a well-publicized (although not security-relevant) processor implementation flaw.

tem, it is important that they not be overlooked during a security evaluation. Although the problems appear to be fixed in subsequent versions of each processor, it is not clear whether the dearth of reported flaws in more modern processors (e.g., Intel 486, Pentium, non-Intel implementations) represents actual improvement in implementations, the predictable lag between product introduction and problem reports, or simply lesser efforts in making such information available.

The long-term goal of the analysis project is the development of a test suite to exercise 80x86 security features and search for additional flaws. Essentially, we are performing an architecture study of the 80x86 protection mechanisms and a penetration testing effort. Sections 5 and 6 review related work and discuss our plans.

2 Background

A high-assurance secure computing system (such as one intended to satisfy the requirements of the Trusted Computer System Evaluation Criteria (TCSEC) [Ncsc85] at B2 or above) must be able to enforce security policies correctly and reliably, even while under hostile attack. Future versions of the system—developed in accordance with appropriate configuration management procedures—must continue to enforce those policies reliably. Moreover, the protection mechanisms that support these properties must be carefully structured and well-defined for evaluation purposes. These fundamental aspects of secure systems form the basis for design, development, criteria, and evaluation worldwide.

In general, however, much more effort is applied to assurance for a system's software components than for hardware components. Designers and evaluators tend to concentrate on the Trusted Computing Base (TCB) software (e.g., the operating system); hardware is assumed to operate securely if used correctly (this assumption is often implicit; [Gutt90] points out the need to verify such assumptions). Any hardware-related effort is primarily to ensure that software makes correct use of documented hardware features. This imbalance in the depth of hardware versus software analysis has been satisfactory[5] in the past, but is increasingly difficult to justify.

2.1 Trusted System Hardware Dependencies

Although software may receive greater attention, hardware components are clearly critical to a system's security. Indeed (but for a few specialized exceptions), all hardware components are part of a system's TCB, and the hardware security mechanisms are an integral part of the Reference Validation Mechanism (RVM).

[5] At least, satisfactory to the extent that most known security exposures involve flaws in software rather than hardware. However, it is unclear that this situation results from an actual absence of hardware flaws; other factors may be at work.

Generally the Central Processing Unit (CPU) is most important. This component, which may include a distinct Memory Management Unit (MMU), I/O processor, and/or other functional units, is responsible for isolating the TCB from untrusted subjects, and subjects from each other. This may involve mechanisms such as user/supervisor state or rings, address space separation, segmentation and segment protection, page protection, I/O device or address protection, etc. These mechanisms are fundamental to security enforcement—without them, the rest of the TCB could not maintain security.

In addition, the CPU is trusted to be functionally correct; that is, to perform correct computations on behalf of the TCB software. If the CPU operates incorrectly, the TCB software may fail unpredictably. However, such failures are less likely to introduce an exploitable security flaw, and more likely to introduce computational errors that will cause non-TCB software to noticeably malfunction.

Other hardware components—memory, disks, tapes, other peripherals—are also trusted to function correctly in order for the TCB software to operate. Because such components rarely contain security mechanisms, their incorrect operation is similarly less likely to introduce an exploitable flaw[6] and more likely be noticed.

2.2 Evaluation of Hardware Assurance

What is the reason for the traditional focus on software assurance in trusted systems? There are several. Foremost, perhaps, is that this traditional approach "appears to work." There are many well-documented examples of security flaws due to software errors, but few for hardware [Land94].

Another reason is that, as developers and evaluators, we concentrate on what we understand and can affect. Hardware is typically presented as a black box, "a given," on which a secure system must be built. This hardware is often procured from a third party with no incentive (or capability) to provide any details about its implementation. In contrast with the malleable nature of software, hardware cannot be modified easily, and often not at all. Furthermore, modifying and assessing hardware components requires different knowledge and skills than for software; these are rarely found together.

Although comfort and familiarity are difficult to justify scientifically, a third reason is on firmer ground: relative complexity and reliability of software versus hardware. In general, the visible interface to a system's hardware components (such as a CPU) is much simpler than the trusted software interface. Because the interface is simpler, it is easier to test thoroughly, and given the speed of hardware operations, far more testing is possible. Hardware design is a more systematic and disciplined process than software design,

[6] However, data dependencies in peripherals, or undocumented I/O commands where a low-level I/O interface is exposed to untrusted subjects, can introduce serious flaws.

contributing to reliability and general correctness (although there is less focus on isolation of security functions in hardware than for secure software). Together, these properties all contribute to a real distinction in the way that hardware assurance is provided.

Unfortunately, simple interfaces do not necessarily correspond to simple implementations. In software, complexity is considered a prime breeding ground for flaws; the same seems likely for hardware. However, limited visibility into hardware implementations makes it more difficult to judge hardware complexity. Furthermore, because hardware components are often designed with an incomplete (or absent) understanding of security issues, the result may be poorly matched to the needs of a secure system.

2.3 Microprocessor Assurance

The microprocessor has changed the way trusted systems are built. In the 1970's, trusted systems were typically built entirely by one company: the same organization, although not necessarily the same people, produced both hardware and software components. Most hardware implementations were relatively simple—complex hardware was extremely costly to design and build. Security-relevant dependencies between hardware and software components could be addressed within the confines of a single organization. Analysis of, documentation for, or assurance about hardware components could be provided within the organization as required.

In the 1980's, this began to change: increasingly, hardware components became commodities, and organizations that had previously built their own processors and peripherals started to acquire those components from external sources over which they had relatively little influence. Fortunately, at least from an assurance perspective, trusted system development lagged this trend, and continued to rely either on proprietary hardware or on relatively simple commodity hardware.

In the 1990's, however, simplicity of hardware components is no longer a given. Processors in particular have become far more complex and less expensive, and trusted systems are migrating to those platforms. In particular, the architectural security features of the 80x86 processors and the prevalence of the 80x86-based PC-compatible architecture as a development and delivery platform has led many developers to target that environment. Table 1 shows all products available in mid-1994 that have been evaluated for TCSEC class B2 or better by the NSA's TPEP program.

Of those, all but XTS-200 is hosted on an 80x86 architecture; however, its follow-on product, XTS-300, is claimed to be targeted for B3 evaluation and runs only on 80x86 platforms. Additionally, Trusted Information Systems (TIS) claims its Trusted Mach [Sebe94] project is targeted for B3 evaluation and employs the Intel 486 as its reference platform, and there are several products in development that also depend on these processors, including several targeted for B1 or below. Clearly, the 80x86 is the processor of choice for trusted systems in the 1990's—and if we were to include the list of systems accredited for multi-level operation, this list would expand significantly.

Unfortunately, this hardware cannot safely be dismissed as "simple" or not in need of in-depth analysis. A relatively simple interface can hide vast implementation complexities—and the 80x86 interface is far from simple. For example, the Pentium contains approximately 3.1 million transistors [Int94]. Its instruction processing and pipeline architecture are extremely complex. The popularity of these processors also raises the concern that a flaw affecting one trusted system could easily affect others.

2.3.1 Assurance By Design and Analysis

Typically, software developers provide assurance by using top-down, modular design, minimizing complexity, structuring the development process, and documenting the design and implementation. Similarly, evaluators assess assurance by examination of the implementation and of the development evidence and documentation. These techniques correspond directly to modern software engineering practice, and involve information that is readily understood by both developers and evaluators.

Unfortunately, these techniques do not apply well to hardware. The first problem is that details of hardware design, structure, documentation, and so forth are generally not available to evaluators—if, indeed, they exist at all. Even in the mainframe era, this information was difficult to obtain within the confines of a single organization. For modern microprocessors, it is effectively impossible—the processor supplier is often unrelated to the system developer, and in any case considers the design details extremely proprietary. Although in-depth accurate hardware design documentation is more likely to exist today, it is simply not available in the context of a trusted system evaluation.

The second problem is that, even if hardware design documentation were available, it would be of limited utility to developers and evaluators whose skills lie largely in the software world. Although for simple processors a software-oriented evaluator may be able to perform an informal assessment of hardware security (see section 5.1), it is impractical to gain the same degree of understanding as one would for software. For the complex implementations of today's microprocessors the situation is much more difficult.

Formal assurance methods represent another approach; this is discussed further in section 5.3.

2.3.2 Assurance by Testing and Exposure

The other primary technique for gaining assurance is testing. Hardware components support testing bet-

Trusted Product	Target Rating	Base Architecture
Boeing MLS LAN	A1	80x86 multiprocessor
Gemini Trusted Network Processor	A1	80x86 PC or multiprocessor
HFSI XTS-200	B3	Bull DPS-6 (proprietary)
Verdix VSLAN	B2	80x86 custom board
TIS Trusted Xenix	B2	80x86 PC

Table 1: Hardware Platforms of High-Assurance Trusted Products

ter than software, so this is relatively more effective. Testing, however, must be directed. It is a common fallacy that the exposure of a system to "millions of users" ensures that it is secure. In fact, this technique is of limited value, even for ensuring that a system is reliable—user acceptance may instead reflect a tolerance of failure, and is in any case relevant primarily to the most heavily-used features of the system.

The more subtle problem with exposure testing for security is that security is the *absence* of undesired behavior. During normal system operation, a user is likely to notice when a function does not behave as advertised, because application programs *use* those functions. One is much less likely to notice that a security policy has *not* been enforced. A correctly operating application program simply will not attempt operations that the security policies would prohibit—there can be no meaningful dependency on such operations. Thus, the exposure of the 80x86 processors to millions of PC users tells us little about the soundness of their security mechanisms.

Not all PC users even use the protection mechanisms (MS-DOS users do not). Of those who do, the vast majority do not depend on the mechanisms to *enforce* security policy. Of that tiny minority that *do* expect security enforcement, few run hostile programs that would attempt operations that the hardware security mechanisms would prevent. Directed testing of the protection mechanisms seems essential, regardless of the total size of the user base.

While directed testing is clearly necessary, it must contend with problems such as undocumented features[7]. A processor with undocumented security-relevant features can undermine the TCB software's[8] best efforts at policy enforcement. Although it is relatively easy to analyze software to determine all the functions it can perform, without visibility into the implementation, this is very difficult for a hardware component—and testing is not an effective substitute.

[7] The Pentium processor brings a new aspect to this problem: it incorporates some functions that are *explicitly* undocumented. These are described in an "Appendix H" to [IntP5] that is available only under strict non-disclosure protection. It is claimed in [IntP5] that these functions only provide optimizations for certain TCB software operations, although without documentation, they cannot be fully analyzed for architectural pitfalls.

3 Architectural Pitfalls

This section describes several pitfalls in using the 80x86 architecture to build secure systems. These result from apparent design oversights and/or unexpected interactions among processor features. In some cases, performance optimizations provide unexpected avenues for information flow.

The 80x86 architecture includes many mechanisms for constructing secure systems:

- Multiple operating modes: Protected, Virtual-8086, Real, System Management Mode (SMM)
- Segment protection: Descriptors, Program and Segment Privilege Levels (PL), Access Modes
- Special segments: Gates, etc.
- Multitasking: Task State Segments (TSSs), Task Switching
- Paging: Address Space Management, Per-page Protection
- I/O protection: I/O Instructions, I/O Permission Bitmap
- Miscellaneous: Control Flags, Debug Registers, CPU Identification, etc.

The details of these mechanisms can be found in [Int386, Int486, IntP5].

The pitfalls discussed here apply only to features (such as the unprivileged instruction set) that are directly visible to unprivileged programs, or that a TCB might virtualize to make indirectly available (such as the debugging registers). Features that would necessarily be used solely by the TCB (such as page tables) are not addressed, since those resources must be entirely in the TCB's control.

Some pitfalls represent covert channels: they permit one subject (process) to perform an operation that is detectable by another subject, in a way that could violate a system's rules for information flow. These are particularly interesting because some were discovered during the course of a high assurance TPEP evaluation, and may be applicable to other products. Fortunately, countermeasures were found to close the flows.

They were not previously detected because the relevant processor features were not represented in the system specifications. The others are simply features whose rules for safe use are more subtle than might be expected from an initial study of the architecture.

3.1 TS Flag Information Flow

The 80x86 processors have a logically distinct Floating Point Unit (FPU) with its own register set and context. Because not all programs need or use the FPU, it would be inefficient to save and restore the FPU contents at every task switch. Therefore, there is a mechanism to help minimize FPU saves and restores: the Task Switched (TS) flag.

The TS flag works as follows: whenever a program executes a FPU instruction, if the TS flag is set, a hardware-detected exception occurs, invoking the TCB. The TCB then has an opportunity to see whether the current process is the one that was last using the FPU. If so, it clears the TS flag and restarts the instruction immediately. If not, it must save the FPU context, re-load the FPU from the current process's FPU context, then clear the TS flag and restart the instruction. The TS flag is set automatically by the hardware whenever a task switch (i.e., loading the processor state for some process) occurs.

This would be fine except that the TS flag is visible outside the TCB. Although it is located in a restricted control register, an unprivileged instruction (Store Machine Status Word, `SMSW`) makes the TS flag visible. The following scenario illustrates how the TS flag could be used as a signaling mechanism, and applies to all 80x86 processors (as well as the 80286).

Figure 1 illustrates one example signaling scenario that could transmit a binary signal (other more efficient forms of this flow exist as well). The scenario assumes that the sender S and receiver R share a single processor, with no other processes active on the processor. It is also assumed that a mechanism exists allowing S to sleep for a specified period of time, and preempt R when this period is completed.

To Signal 0:

- R begins the cycle by executing a FPU instruction, clearing the TS flag.
- R enters a loop for X iterations. During this period, no swapping occurs and the TS flag remains clear.
- Upon completion of the loop, R polls the TS flag using `SMSW`, and determines that no preemption occurred. A clear TS flag indicates the sender has transmitted a "0".

To Signal 1:

- S begins the cycle by requesting an interrupt to awaken it M clock ticks from now. M is calculated such that the sender will be awakened during R's $(X-N)^{th}$ loop iteration.
- R then executes a FPU instruction, clearing the TS flag.
- R enters a loop for X iterations.
- (X-N) iterations into R's loop, the interrupt occurs, causing S to be awakened. Upon exchanging S for R, the task switch sets the TS flag.
- S immediately sleeps, allowing R to swap back in. At this point, R is unaware that it had been pre-empted.
- Upon completing the loop, R polls the TS flag. A set TS flag indicates the sender has transmitted a "1".

Although a TCB-provided timer facility is assumed for the example, this is not a necessary aspect of the channel. For instance, in a multiprocessor system, the real sender S, running on processor A, could simply send interrupts or IPC messages to its proxy S′ that is sharing processor B with R, causing S′ to wake up (and thus task switch) or not.

One difficulty in attempting to detect this flow with a Formal Top-Level Specification (FTLS) and/or code analysis is that the setting of the TS flag occurs internally to the processor. This state change is thus often omitted from analysis as an implementation detail—which has indeed been observed in one case.

Interestingly, beginning with the i386, Intel moved the TS flag into the privileged CR0 register (it had previously been located in the 80286 Machine Status Word, which was removed in the i386). Intel recommends against further use of the `SMSW` instruction, and instead recommends use of the privileged `MOV CR0` instruction. Unfortunately, compatibility with old (80286-based) programs requires that `SMSW`, and thus the value of the TS flag, remain available to unprivileged programs.

3.2 FPU Context Timing Channel

The previous scenario illustrates an information flow based on visibility of the TS flag with `SMSW`. However, even if the TS flag were not visible, its state could potentially be sensed by the duration of a FPU instruction. If the TS flag is set, an exception will occur and be handled by the TCB before the FPU instruction completes. This is likely to take considerably longer than if the TS flag were clear. This delay can be observed as greater execution time for the FPU instruction.

Additionally, the Intel scheme for handling numeric exception errors is to handle the error in the context of the next FPU instruction, regardless of which process executes the next instruction. Thus if process S performs an erroneous FPU instruction and immediately

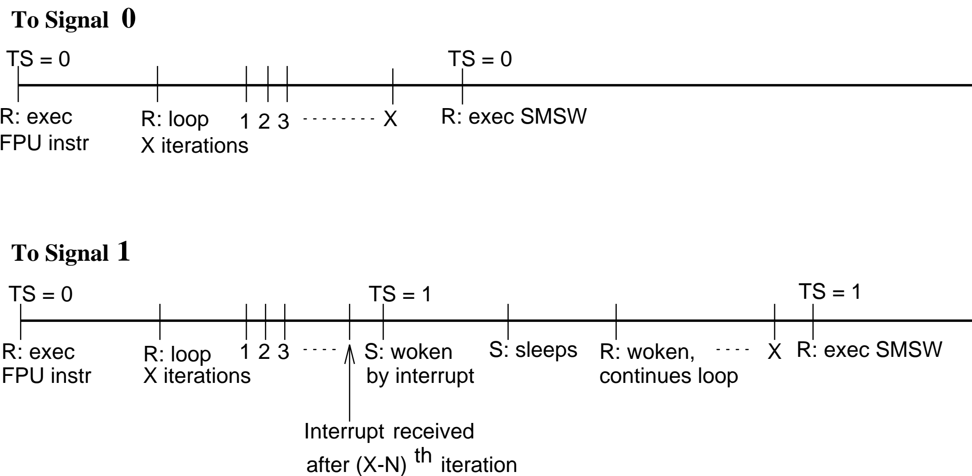

Figure 1: TS Flag Covert Channel Scenario

swaps out, process R can sense this by performing a FPU instruction and observing the delay from the FPU context swap, plus the delay from handling the numeric exception.

Fundamentally, these flows are caused by the FPU context-saving optimization itself, not by visibility of the TS flag. Thus, a full remedy requires making the TS flag's effects, as well as its value, invisible to unprivileged programs.

3.3 Segment Accessed Bit

The "accessed" bit in a segment descriptor is visible through the Load Access Rights (LAR) instruction. This bit is set whenever the segment's contents are accessed, and is reset only by TCB software.

If two processes share access to a segment descriptor (for instance, a read-only data segment in the Global Descriptor Table (GDT)), a sending process can read from the segment and in doing so, signal to a receiving process that is waiting for the accessed bit to change. In practice, this is unlikely to be a serious problem because the bit, once set, could be reset only by TCB software, and because the number of possible GDT entries is fixed. However, it is clearly a flaw: an operation (performed by the sender) whose semantics clearly imply "read" actually performs a "write," and the result of that "write" is visible to the receiver without restriction.

A simple countermeasure is to ensure that the "accessed" bit of each descriptor is set when the descriptor is created. This, however, would effectively disable the feature, and render it unavailable to the TCB for use in segment management.

3.4 Other Segment Attributes

A similar situation to the segment-accessed bit exists for other segment attributes. There is an "available for software" bit that is also visible with LAR, as well as a "present" bit; the TCB's use of those bits must be designed to avoid possible flows. Another attribute (segment limit) is also visible through an instruction (Load Segment Limit, LSL), and could be used to determine whether a segment had been extended by a reference beyond its earlier limit. A third potential problem comes from access rights themselves, visible both with LAR and the access-checking instructions VERify Read (VERR) and VERify Write (VERW): if a system uses access rights to implement demand segmentation, a program can determine whether a segment is actually writable. This could yield different results when checking writability than when attempting a write, as the latter would be made to work by the system's fault handler.

As with segment-accessed, none of these appear to present great risk, and they seem unlikely to be exploitable in real systems. However, they are examples of a design that is problematic for security in that it makes visible real attributes that should be virtualizable by a TCB. Because the applicable instructions are available without restriction (except for the target segment's PL), the TCB has no effective means of controlling their use, and must instead be designed to eliminate their potential misuse.

The effect of these instructions is subtle: although it is clear that the Page Directory Table and GDT themselves must be protected, it is not as obvious that the parts of descriptors that can be read (but not modified) through these instructions can present a security problem

3.5 Page Access (In)Visibility

The attribute-visibility pitfalls appear only at the segment level: there are no corresponding operations for interrogating page attributes, so page management can be truly invisible outside the TCB.

Unfortunately, this means that, based on segment attributes, the access-checking instructions (VERR and VERW) can report that an address is readable (or writable) when it actually is not. A TCB that relies on page-level protection to enforce access control cannot use VERR and VERW to validate parameter addresses—even though a segment may be accessible, pages may be individually protected. These two instructions were provided to help OS developers avoid access problems caused by invalid parameter addresses. Unfortunately, they are unreliable for systems using page-level protection, because they could lead the TCB to conclude incorrectly (and insecurely), that its caller has appropriate access to a protected page.

3.6 Internal Register Visibility

Several internal registers (those designating the local, global and interrupt descriptor tables; LDT, GDT, and IDT respectively) are visible with unprivileged instructions. A program can issue the (Store LDTR, SLDT) instruction to determine the current location of its LDT. If LDTs are ever moved or reassigned, perhaps in response to contention caused by other subjects, SLDT makes such changes visible, thus creating a potential information flow. Because the GDT and IDT are system-wide tables, their locations are less likely to be useful for covert channels.

Like segment attributes, the LDTR represents information that should be virtualizable by the TCB—or simply not accessible at all. Because the SLDT instruction is unprivileged, the TCB must be designed to preclude use of the LDTR as a covert channel.

3.7 Debug Register Values

Because the Debug Registers (DRs) provide a powerful facility for program debugging, a general-purpose system may make them visible to unprivileged subjects. The DRs are accessible only at PL 0, so they must be virtualized and kept as part of a process's context. However, this is not sufficient protection: the *values* set in DRs must also be validated by the TCB, to eliminate the potential for interference with TCB operation.

The breakpoint addresses are linear addresses, and so must be calculated by the TCB relative to the process's segments.[8] The control values must be validated to avoid introducing undefined values into the DRs, or setting values in undefined DR fields. In practice, this should not be difficult.

3.8 Time Stamp Counter

The Pentium processors support a time stamp counter (TSC) register that provides a count of machine cycles since the processor was reset. This allows extremely high-resolution timing of program activity and is useful for performance measurement and real-time programs. Although this is not fully documented in non-proprietary parts of [IntP5], that manual contains sufficient hints to determine how to understand the feature and make it available. In addition, [VanG94] provides a more detailed description.

High-resolution timing, however, is also the key to efficient exploitation of covert timing channels [Hu91]. Fortunately, the Pentium has a control flag that makes the Read TSC (RDTSC) instruction privileged; by making RDTSC privileged, a TCB can virtualize the TSC to reduce its effectiveness for covert channels. As with any high-resolution clock, the TSC must be either virtualized or eliminated entirely to reduce the covert channel threat.

3.9 Performance Counters

The Pentium has a set of Model-Specific Registers (MSRs), most of which are undefined in [IntP5]. They are, however, defined to be accessible only at PL 0, thus reserving them to the TCB.

Recently, descriptions of some of these registers have been published [Math94], based on analysis of an Intel-developed performance monitoring package. These MSRs count internal processor events, such as cache misses, pipeline activity, etc., and can be used for characterizing program performance.

Although they are not directly accessible outside PL 0, a TCB could (because they are useful to application programmers) provide virtual access to these MSRs. As with the TSC, however, this requires care to reduce the threat from covert channels. Unlike the TSC, which measures an external clock, some performance MSRs can measure effects of other concurrent activities (e.g., cache snoops by another processor) or of previous activities (e.g., cache hits and misses due to cache activity by a previous subject). Earlier work [Wray91] has shown how to exploit such mechanisms based on measurement against a time reference, but MSRs provide a direct indication.

3.10 Cache and TLB Timing Channels

As mentioned in the preceding scenario, caches present potential for covert timing channels. Even without MSRs for direct measurements of cache activity, cache hits and misses can be detected strictly from instruction timing, as described in [Wray91]. To eliminate these flows, caches must be managed. This can reduce their efficiency considerably, depending on cache architecture, as it introduces otherwise unnecessary cache flush and invalidation activity.

[8] In a system that relies on paging and page protection for all its separation, and provides a subject with one code and one data segment covering the entire linear address space, the breakpoint address need not be translated—however, it still must be validated to ensure that it does not permit breakpoints to be set for TCB addresses.

The Intel 486 and later processors include caches for data and instructions, as do some of the non-Intel 80386-family processors. In addition, all 80x86 processors have a Translation Lookaside Buffer (TLB) that, although smaller than the data caches, has potential for use as a covert timing channel.

3.11 Undefined Values

Most 80x86 computational instructions are defined explicitly to return "undefined" values for arithmetic flags, presumably to aid implementation optimizations. Although it seems extremely likely that the actual values of the flags are derived deterministically from legitimately accessible information,[9] the architecture does not require it, and it is conceivable that they might represent the results of some processor activity performed by a previous subject and thus usable as a covert storage channel.

A few unprivileged instructions (for example, `LAR`) return values in which some bits are explicitly undefined. Again, it seems extremely likely that those bits are derived from legitimately accessible information, but that is not required by the specification.

4 Reported Implementation Errors

The 80x86 processors, particularly the early versions of the Intel 386 and Intel 486, have had a history of implementation errors reported in the computer trade press. One such widely reported error involved incorrect results from 32-bit multiplication with certain operand values in some Intel 386 "B-step" processors; Intel actually undertook additional testing and, until the problem was fixed, provided specially-marked versions of these processors in which the error did not occur. An even more widely reported processor error, the Pentium `FDIV` flaw, led to the replacement of faulty processors and may lead to significant changes in how such errors are handled in the future.

All of these widely-reported flaws seemed to involve errors that, while they might cause an application program or possibly a TCB operation to operate incorrectly, did not appear to translate into exploitable security flaws. A more in-depth search for reported flaws, however, turned up some that clearly did involve the security architecture, as well as numerous others that did not.

These reports also suggest several general properties of flaws:

1. It takes a while for *any* flaw reports to become public. This appears at least partly due to the understandable reluctance of system and processor manufacturers to announce flaws in their products.[10] Occasional user reports of application malfunctions are not of much interest to the trade press, and in any case it may take considerable engineering effort to determine that a malfunction is caused by a hardware flaw.

2. Flaw reports (and, presumably, flaws) are much more common for early versions of the processors. Obviously: manufacturers are strongly motivated to fix these problems while their effect on users is limited. Often, flaw reports are not available until long after the flaws have been fixed.

3. Flaw reports concentrate on processor functions that are heavily used in common application environments, but are not limited to those functions. Again, this is no surprise: if a processor feature is not used, it is unlikely that user exposure will result in discovery of an error in its implementation. If anything, it is a little surprising that some of the reported flaws deal with obscure or pathological uses of processor features—it seems clear that some of these reports derive from explicit testing rather than accidental discovery.

Overall, it does not seem safe to generalize from these properties. Without visibility into the proprietary development procedures of processor and system manufacturers, we cannot reach any conclusion about, for example, why there are so few distinct reported flaws in the Pentium. Although one Pentium flaw (in floating point divide) has been extensively reported, few other reports have surfaced. It is difficult to know whether this is due to the customary reporting lag, an actual absence of flaws, or simply a lack of available reports. It is similarly difficult to determine whether the little-used processor features are more or less likely to harbor flaws (on the one hand, they are less exercised, so errors may not have been discovered; on the other hand, they are less likely to have a complex and optimized implementation, so errors may not have been introduced).

4.1 Reported Flaws

We have identified 102 reports of flaws affecting various versions of 80x86 processors from the descriptions in [Agar91, Humm92, Turl88, Mor93].

In many cases, the reports do not identify specific processor versions, but refer to "early" or "some" versions; where there are multiple reports of the same flaw, it is counted in the category with the most specific version information. It seems generally safe to assume that a flaw reported in one version is also present in the earlier versions; however, there are a few reports of new flaws introduced by new versions. It is also not clear the extent to which non-Intel processors exhibit any of these faults. Some AMD processors, which share microcode with the Intel versions, would

[9] There is a parallel here to [Karg74], in which an undocumented instruction was discovered on the GE-645 that accessed an internal processor register of no apparent significance.

[10] In response to the Pentium problems publicized in late 1994, Intel has announced a policy of making "errata sheets" available on a more public basis than in the past.

presumably share any microcode-based flaws. Others, such as Cyrix processors could introduce flaws not found in Intel or AMD versions [Meth94].

We classified software flaw reports as follows:

Security This is a flaw that could possibly be exploited directly by an unprivileged program to violate some hardware-enforced access control or restriction. These do not, however, include any misbehavior resulting from inappropriate setup of TCB data structures, since that is presumably under the TCB's control.

Denial of Service This is a flaw that could apparently be used by an unprivileged program to "hang" the CPU so that it can no longer operate. Again, these do not include problems due to incorrect initialization of TCB data structures.

Other These are all the other flaws, such as those causing incorrect computational results, those where an incorrect exception is reported, or where incorrect values in a TCB data structure cause some anomalous behavior. Although these flaws could potentially affect correct operation of TCB software, they are not considered security-relevant because the effect is so indirect.

In addition to the instruction flaw reports, there are several reported flaws concerning incorrect bus or cache operation. Because these can be exercised only from outside the processor (e.g., by I/O devices) and not by instruction sequences, they were not considered in this analysis.

Table 2 characterizes the 102 reports of distinct flaws that we have encountered. Of these flaws, 17 are security-relevant. From the reports on which this table is based, it appears that it would be generally safe[11] to construct a trusted system with any Intel 386 processor later than step B1, and any Pentium. It must be emphasized, however, that this is based only on public reports of flaws: no attempt has yet been made to verify these reports or to determine whether they have been fixed in later processor versions. In addition, as discussed above, it is not clear whether the lack of reported security flaws in the Pentium corresponds to a lack of actual flaws or simply a lack of available reports.

4.2 Security Flaws

This section describes potential security flaws that have been reported for various versions of the Intel 386 and Intel 486 processors. The descriptions in this and section 4.3 are necessarily terse and dependent on a detailed knowledge of the 80x86 instruction set. A reference to a more detailed description is associated with flaws 1-7. Flaw 8 was independently confirmed by our group.

We considered a flaw security-relevant if it appeared to have any effect on security-critical data within the processor. For example, anything that results in an incorrect linear address being generated would be security-relevant. Similarly, any mis-handling of descriptors, incorrect PL checks, or situations where protection mechanisms do not function properly, would be considered security-relevant. We did not attempt to construct detailed exploitation scenarios for these flaws because those would necessarily be dependent on an operating system's architecture. Although in some cases it is difficult to see how a security-relevant flaw might be exploited, it is even more difficult to show that exploitation would not be possible, and we chose to err on the conservative side in our assessments. The existence of any known flaws, and the possibility that others might be present but unknown, argues strongly for avoiding the affected processor versions in any trusted system.

1. The `LSS` instruction is used to load the Stack Selector with data from memory. `LSS` should verify that the RPL of the selector equals the DPL of the current code segment. The A1 step of the Intel 386 fails to perform this check. [Turl88]

2. When popping a segment selector from the stack, `POP` should verify that the segment is accessible. The A1 step of the Intel 386, however, performs this check incorrectly, allowing `POP`s for otherwise inaccessible segments. [Turl88]

3. Interrupts that occur in Virtual 8086 mode are handled in Protected mode. Upon returning from the interrupt, the B0 step of the Intel 386 fails to truncate the stack offset address to 16-bits. [Turl88]

4. When transferring control from a 16-bit code segment to a 32-bit code segment via a task gate or gate call, but without a PL change, the B0 step of the Intel 386 truncates the EIP to 16-bits. [Turl88]

5. The Translation Lookaside Buffer (TLB) is sometimes used for I/O addresses 1000h and up, and also coprocessor addresses; neither of these should ever be translated. [Turl88]

6. Prefetching may fetch otherwise inaccessible instructions in Virtual 8086 mode (unspecified version). [Humm92]

7. The bits of the I/O Permission Bitmap (IOPB) correspond to individual byte addresses in the I/O address space. The D0 step of the Intel 386 permits access to certain addresses prohibited by the I/O bitmap: if a 4-byte access is performed, only 3 of the 4 relevant bytes are checked. [Mpr93]

[11] The security flaw for the Intel 386 D0 step involves incorrect interpretation of the I/O permission bitmap. A system that does not use that feature would not be affected by this flaw; even systems that do use it would generally not be adversely affected. However, to be safe, one should use a processor later than the D0 step if the I/O bitmap is used.

Processor and Revision	Distinct Reported Flaws		
	Total	Security	D. of S.
Intel 386 A1 step	28	2	6
Intel 386 B0 step	12	2	0
Intel 386 B1 step	15	1	2
Intel 386 D0 step	3	1	0
Intel 386 "some versions"	19	1	1
Intel 386 "all versions"	1	0	0
Intel 486 "early versions"	6	0	0
Intel 486 "some versions"	8	0	0
Intel 486 A-B4 steps	3	0	0
Intel 486 A-C0 steps	2	0	0
Intel 486 "all versions"	2	1	0
Cyrix 486	2	0	0
Intel Pentium	1	0	0

Table 2: Flaws in 80x86 Microprocessor Revisions

8. **INVLPG, INVD** and **WBINVD** are privileged instructions introduced in the Intel 486 to invalidate TLB entries, invalidate cache, and flush and invalidate cache, respectively. Although they are designated privileged instructions, they operate successfully when executed from a non-privileged state (PL1-PL3).

4.3 Denial of Service Flaws

This section describes flaws that have been reported to cause various versions of the Intel 386 processor to hang (thus, potentially, denying service to other users). The effect of all these flaws is identical: using these non-privileged instructions, a program can effectively halt the processor—and the system

1. **LAL, LSL, VERR, VERW** for a null (zero) selector (A1 step) [Turl88]
2. Unaligned 16-bit selector operand loaded with **MOV, LDS, LES, LFS, LGS, LSS** (A1 step) [Turl88]
3. **FAR JMP** or **FAR CALL** with last two bytes beyond segment limit (A1 step) [Turl88]
4. **FAR JMP** or **FAR CALL** with last two bytes across a page boundary (A1 step) [Turl88]
5. **BSF** with memory operand that should generate page or GP fault (A1 step) [Turl88]
6. Some undefined FPU opcodes (A1 step) [Turl88]
7. **LAR, LSL, VERR, VERW** for inaccessible selector depending on contents of instruction pre-fetch queue (B1 step) [Turl88]
8. FPU instruction split across page boundaries with second byte not accessible (B1 step) [Turl88]
9. FPU operand that wraps around the end of a segment but straddles an inaccessible page (unspecified version) [Turl88]

5 Related Work

There have been relatively few published studies of security assurance provided by hardware components. Most address architectural, not implementation, aspects.

5.1 Hardware Analysis

In [Glig85], there is a description of the security analysis of the relatively simple security-enhanced Honeywell Level-6 minicomputer used in the A1-evaluated SCOMP system The analysis was performed relative to a specification of security properties, treating each instruction as a security-preserving transition.

The Multics B2 evaluation [Mult85] included an informal analysis of security mechanism implementations in all hardware components. This was performed in 1984 by one of the authors (Sibert), based on interactive guidance from hardware design engineers. For the CPU, this analysis was based on walk-throughs of the security-critical parts of the CPU logic design at the gate level (the Multics CPU is not microcoded). For the microcoded I/O controllers, the critical parts of microcode were examined.

There are several studies (e.g., [Baue84]) about architectural suitability of specific processors for building secure systems, and many more about hardware security features in general. These generally deal with high-level issues (e.g., are there enough domains, is process switching too expensive?), rather than with the sort of low-level details that yield the architectural pitfalls discussed in this paper. In [McAu92], there is a discussion of maintaining hardware assurance when modifying a trusted product, but it too addresses only high-level architecture aspects.

5.2 Hardware Penetration Testing

An important work is [Karg74], which describes an extensive test effort directed at finding hardware flaws in the GE-645 Multics CPU. This culminated in a program (the "subverter") which achieved its goal by demonstrating a combination of instructions and indirect addressing that bypassed access checking on one of the segments involved. The subverter was also run for long periods in the hope of encountering random failures to perform access checks, but none were detected. A similar program was developed in the early 1980's to exercise the Digital VAX 11/780 architecture [Karg94].

5.3 Hardware Verification

There have been some attempts to use formal methods to verify the correct implementation of a processor, such as [Croc88, Cull89, Hunt87, Joyc88, Levy92, Wind90]. These efforts were directed at overall correctness, a superset of security correctness. However, these efforts dealt with processors that are far simpler than modern general-purpose microprocessors, and generally employ simplified specifications of these processors. It is not clear that such techniques are scalable to modern microprocessors. Furthermore, the amount of time expended in these efforts has apparently discouraged chip manufacturers from investing in them except for some small research projects, the authors are aware of no current attempts to integrate formal verification into a commercial microprocessor development cycle.

6 Future Directions

Thus far, this project has identified architectural pitfalls and categorized flaw reports in one microprocessor architecture. Our findings point out the utility—indeed the necessity—for the closer examination of microprocessors in high-assurance secure system development.

Next, we intend to perform a thorough penetration style test of the 80x86 processors. Such testing is limited to showing the *presence* of flaws, not their absence, but we believe that even if testing discovers no flaws, it will provide additional empirical support for trusting the processors—and if security flaws are discover, it will be extremely important for secure system developers to avoid them

In addition, we will develop tests to exercise already-reported flaws, at least those for which we have enough detail to construct a plausible exploitation scenario. This effort may be hampered by the unavailability of flawed processor versions on which to run the tests, but the goal is to have tests that can show easily whether a particular processor exhibits any of the known security flaws, rather than relying on ambiguous associations of flaw reports with processor versions.

We will build a general purpose framework for test development to aid future penetration test efforts. The framework will support easy creation of protected mode environments, allowing customization of the framework's environment to correspond to the environments used by different secure systems, which differ in their use of various processor mechanisms (e.g., use of segments and paging). The framework will also support automated test case generation.

Currently, our penetration effort is limited by availability of information about the processors. In traditional penetration testing efforts, evaluators have complete access to internal design and implementation information about the system. Here, we are using only public information. A more thorough and cost-effective test could be done if design information were available. Despite these limitations, we are optimistic; we can only hope to be as successful as the analysis reported in [Karg74], which was conducted under similar conditions.

The planned tests are static: they will set up execution conditions in a stand-alone environment and observe the results. It would be a valuable extension to run these tests under more stressful conditions, such as might arise in a multiprocessor system the interaction of externally-caused cache flushes, bus interactions, etc., might reveal additional problems.

Lastly, it is not clear whether the problems identified here demonstrate greater weaknesses within the 80x86 family (perhaps due to its complex architecture) than for other processors, or that they merely reflect the popularity and exposure of the 80x86 relative to its competitors. During our literature survey for microprocessor implementation, we found the preponderance of flaw reports concerned 80x86 processors, while other processor families (e.g., SPARC, MIPS, Motorola 68000) had few or no reported flaws.

On the other hand, the absence of numerous flaw reports does not necessarily translate into a secure component. A recent examination of the Motorola 88110 processor, for example, revealed that its floating point status register provides a flag analogous to the 80x86 TS flag (see section 3.1) that is visible to user mode processes. Particularly if our tests prove successful in identifying additional security-relevant problems with 80x86 processors, we may extend our analysis to other processor families.

Acknowledgments

Conversations with Daniel Tapper led to the identification of the Floating Point Unit timing channel discussed in section 3.2.

References

[Agar91] Rakesh K. Agarwal, 80x86 Architecture and Programming, Prentice Hall, 1991.

[Baue84] R. K. Bauer, R. J. Feiertag, B. L. Kahn, and W. F. Wilson, Security Concepts for Microprocessor-Based Key Generator Controllers,

Sytek Corporation TR-84009 (contract MDA904-82-C-0449), April 1984.

[Croc88] S. D. Crocker, E. Cohen, S. Landauer, H. Orman, Reverification of a Microprocessor, Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, pages 166-176, 1988

[Cull89] W. Cullyer, *Implementing High Integrity Systems: The Viper Microprocessor,* IEEE AES Magazine, May 1989.

[Glig85] Virgil D. Gligor, *Analysis of the Hardware Verification of the Honeywell SCOMP,* Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, pages 32-43, 1985.

[Gutt90] Joshua D. Guttman, Hai-Ping Ko, *Verifying a Hardware Security Architecture,* Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, pages 333-344, 1990.

[Hu91] Wei-Ming Hu, *Reducing Timing Channels with Fuzzy Time,* Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, pages 8-20, 1991.

[Humm92] PC Magazine Programmer's Technical Reference: The Processor and Coprocessor, Robert L. Hummel, Ziff-Davis Press, Emeryville, CA, 1992.

[Hunt87] W. A. Hunt, The Mechanical Verification of a Microprocessor Design, In D. Borrione, editor, From HDL Descriptions to Guaranteed Correct Circuit Diagrams, North-Holland, 1987

[Int94] Microprocessors, Volume III: Pentium Processors, Order Number 241732, Intel Corporation, Santa Clara, CA, 1994.

[Int386] Intel386 Microprocessor Family Programmer's Reference Manual, Order Number 230985, Intel Corporation, Santa Clara, CA, 1989.

[Int486] Intel486 Microprocessor Family Programmer's Reference Manual, Order Number 240486-002, Intel Corporation, Santa Clara, CA, 1992.

[IntP5] Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual, Order Number 241430-001, Intel Corporation, Santa Clara, CA, 1994.

[Joyc88] Jeffery J. Joyce, Formal Verification and Implementation of a Microprocessor, In G. Birtwistle and P. A. Subrahamanyam, editors, VLSI Specification, Verification, and Synthesis, Kluwer Academic Press, 1988

[Karg74] Paul A. Karger and Roger R. Schell, *Multics Security Evaluation: Vulnerability Analysis,* Air Force Electronic Systems Division ESD-TR-74-193, Volume II, June 1974.

[Karg94] Paul A. Karger, Personal communication, October 1994.

[Land94] Carl E. Landwehr, Alan R. Bull, John P. McDermott, William S. Choi, *A Taxonomy of Computer Program Security Flaws, with Examples,* Computing Surveys, Volume 26, Number 3, pages 211-254, September 1994.

[Levy92] Beth Levy, Ivan Filippenko, Leo Marcus, Telas Menas, *Using the State Delta Verification System (SDVS) for Hardware Verification,* in Theorem Provers in Circuit Design, North-Holland, pages 337-360, 1992.

[Math94] Terje Mathisen, *Pentium Secrets,* Byte Magazine, Volume 19, Number 7, pages 191-192, July 1994.

[McAu92] Noelle McAuliffe, *Extending Our Hardware Base: A Worked Example,* Proceedings of the National Computer Security Conference, Baltimore, MD, pages 184-193, 1992.

[Meth94] David Methvin, *Compatible ... Or Not?,* Windows Magazine, pages 217-220, June 1994.

[Mpr93] Understanding x86 Microprocessors, (A collection of 99 articles from Microprocessor Report), Ziff-Davis Press, Emeryville, CA, 1993

[Mult85] Final Evaluation Report: Honeywell Multics Release MR11.0, National Computer Security Center, NCSC Report CSC-EPL-85/003, Library No. S227,783, June 1986.

[Ncsc85] National Computer Security Center, *Trusted Computer System Evaluation Criteria,* DoD, DoD 5200.28-STD, December 1985.

[Sebe94] E. John Sebes, Nancy Kelem, Terry C.V. Benzel, Mary Bernstein, Eve Cohen, Jeff Jones, Jon King, Michael Barnett, David M Gallon, Roman Zacjew, *The Architecture of Triad: A Distributed, Real-Time, Trusted System,* Proceedings of the National Computer Security Conference, Baltimore, MD, pages 184-193, 1994.

[Turl88] James L. Turley, Advanced 80386 Programming Techniques, Osbourne McGraw-Hill, Berkeley, CA, 1988.

[VanG94] Frank Van Gilluwe, The Undocumented PC, Addison-Wesley Publishing Company, Reading, MA, 1994.

[Wind90] Phillip J. Windley, A Hierarchical Methodology for Verifying Microprogrammed Microprocessors, Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, pages 345-357, 1990

[Wray91] John C. Wray, *An Analysis of Covert Timing Channels,* Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, pages 2-7, 1991.