Hardware aspects of secure computing

by LEE M. MOLHO

System Development Corporation
Santa Monica, California

INTRODUCTION

It makes no sense to discuss software for privacy-preserving or secure time-shared computing without considering the hardware on which it is to run. Software access controls rely upon certain pieces of hardware. If these can go dead or be deliberately disabled without warning, then all that remains is false security.

This paper is about hardware aspects of controlled-access time-shared computing.* A detailed study was recently made of two pieces of hardware that are required for secure time-sharing on an IBM System 360 Model 50 computer: the storage protection system and the Problem/Supervisor state control system.[1] It uncovered over a hundred cases where a single hardware failure will compromise security without giving an alarm. Hazards of this kind, which are present in any computer hardware which supports software access controls, have been essentially eliminated in the SDC ADEPT–50 Time-Sharing System through techniques described herein.[2]

Analysis based on that work has clarified what avenues are available for subversion via hardware; they are outlined in this paper. A number of ways to fill these security gaps are then developed, including methods applicable to a variety of computers. Administrative policy considerations, problems in security certification of hardware, and hardware design considerations for secure time-shared computing also receive comment.

FAILURE, SUBVERSION, AND SECURITY

Two types of security problem can be found in computer hardware. One is the problem of hardware failure. This includes not only computer logic that fails by itself, but also miswiring and faulty hardware caused by improper maintenance ("Customer Engineer") activity, including CE errors in making field-installable engineering changes.

The other security problem is the cloak-and-dagger question of the susceptibility of hardware to subversion by unauthorized persons. Can trivial hardware changes jeopardize a secure computing facility even if the software remains completely pure? This problem and the hardware failure problem, which will be considered in depth, are related.

Weak points for logic failure

Previous work involved an investigation of portions of the 360/50 hardware.[1] Its primary objective was to pinpoint single-failure problem locations. The question was asked, "If this element fails, will hardware required for secure computing go dead without giving an alarm?" A total of 99 single-failure hazards were found in the 360/50 storage protection hardware; they produce a variety of system effects. Three such logic elements were found in the simpler Problem/Supervisor state (PSW bit 15) logic. A failure in this logic would cause the 360/50 to always operate in the Supervisor state.

An assumption was made in finding single-failure logic problems which at first may seem more restrictive than it really is: A failure is defined as having occurred if the output of a logic element remains in an invalid state based on the states of its inputs. Other failure modes certainly exist for logic elements, but they reduce to this case as follows: (1) an intermittent logic element meets this criterion, but only part of the time; (2) a shorted or open input will cause an invalid output state at least part of the time; (3) a logic element which exhibits excessive signal delay will appear to have an invalid output state for some time after any input transition; (4) an output wire which has been con-

*The relationship between "security" and "privacy" has been discussed elsewhere.[3,4] In this paper "security" is used to cover controlled-access computing in general.

From the collection of the Computer History Museum (www.computerhistory.org)

nected to an improper location will have an invalid output state based on its inputs at least part of the time; such a connection may also have permanently damaged the element, making its output independent of its input. It should be noted that failure possibilities were counted; for those relatively few cases where a security problem is caused whether the element gets stuck in "high" or in "low" state, two possibilities were counted.

A situation was frequently encountered which is considered in a general way in the following section, but which is touched upon here. Many more logic elements besides those tallied would cause the storage protection hardware to go dead if they failed, but fortunately (from a security viewpoint) their failure would cause some other essential part of the 360/50 to fail, leading to an overall system crash. "Failure detection by faulty system operation" keeps many logic elements from becoming security problems.

Circumventing logic failure

Providing redundant logic is a reasonable first suggestion as a means of eliminating single failures as security problems. However, redundancy has some limits which are not apparent until a close look is taken at the areas of security concern within the Central Processing Unit (CPU). Security problems are really in control logic, such as the logic activated by a storage protect violation signal, rather than in multi-bit data paths, where redundancy in the form of error-detecting and error-correcting codes is often useful. Indeed, the 360/50 CPU already uses an error-detecting code extensively, since parity checks are made on many multi-bit paths within it.

Effective use of redundant logic presents another problem. One must fully understand the system as it stands to know what needs to be added. Putting it another way, full hardware certification must take place before redundancy can be added (or appreciated, if the manufacturer claims it is there to begin with).

Lastly, some areas of hardware do not lend themselves too easily to redundancy: There can be only one address at a time to the Read-Only-Storage (ROS) unit whose microprograms control the 360/50 CPU.[5,6] One could, of course, use such a scheme as triple-modular redundancy on all control paths, providing three copies of ROS in the bargain. The result of such an approach would not be much like a 360/50.

Redundancy has a specialized, supplementary application in conjunction with hardware certification. After the process of certification reveals which logic elements can be checked by software at low overhead, redundant logic may be added to take care of the remainder. A good example is found in the storage protection logic. Eleven failure possibilities exist where protection interrupts would cause an incorrect microprogram branch upon failure. These failure possibilities arise in part from the logic elements driven by one control signal line. This signal could be provided redundantly to make the hardware secure.

Software tests provide another way to eliminate hardware failure as a security problem. Code can be written which should cause a protection or privileged-operation interrupt; to pass the test the interrupt must react appropriately. Such software must interface the operating system software for scheduling and storage-protect lock alteration, but must execute in Problem state to perform its tests. There is clearly a tradeoff between system overhead and rate of testing. As previously mentioned, hardware certification must be performed to ascertain what hardware can be checked by software tests, and how to check it.

Software testing of critical hardware is a simple and reasonable approach, given hardware certification; it is closely related to a larger problem, that of testing for software holes with software. Software testing of hardware, added to the SDC ADEPT–50 Time-Sharing System, has eliminated over 85 percent of present single-failure hazards in the 360/50 CPU.

Microprogramming could also be put to work to combat failure problems. A microprogrammed routine could be included in ROS which would automatically test critical hardware, taking immediate action if the test were not passed. Such a microprogram could either be in the form of an executable instruction (e.g., TEST PROTECTION), or could be automatic, as part of the timer-update sequence, for example.

A microprogrammed test would have much lower overhead than an equivalent software test performed at the same rate; if automatic, it would test even in the middle of user-program execution. A preliminary design of a storage-protection test that would be exercised every timer update time (60 times per second) indicated an overhead of only 0.015 percent (150 test cycles for every million ROS cycles). Of even greater significance is that microprogrammed testing is specifiable. A hardware vendor can be given the burden of proof of showing that the tests are complete; the vendor would have to take the testing requirement into account in design. The process of hardware certification could be reduced to a design review of vendor tests if this approach were taken.

Retrofitting microprogrammed testing in a 360/50 would not involve extensive hardware changes, but some changes would have to be made. Testing microprograms would have to be written by the manu-

From the collection of the Computer History Museum (www.computerhistory.org)

facturer; new ROS storage elements would have to be fabricated. A small amount of logic and a large amount of documentation would also have to be changed.

Logic failure can be totally eliminated as a security problem in computer hardware by these methods. A finite effort and minor overhead are required; what logic is secured depends upon the approach taken. If microprogram or software functional testing is used, miswiring and dead hardware caused by CE errors will also be discovered.

Subversion techniques

It is worthwhile to take the position of a would-be system subverter, and proceed to look at the easiest and best ways of using the 360/50 to steal files from unsuspecting users. What hardware changes would have to be made to gain access to protected core memory or to enter the Supervisor state?

Fixed changes to eliminate hardware features are obvious enough; just remove the wire that carries the signal to set PSW bit 15, for example. But such changes are physically identical to hardware failures, since something is permanently wrong. As any functional testing for dead hardware will discover a fixed change, a potential subverter must be more clever.

In ADEPT–50, a user is swapped in periodically for a brief length of time (a "quantum"). During his quantum, a user can have access to the 360/50 at the machine-language level; no interpretive program comes between the user and his program unless, of course, he requests it. Thus, a clever subverter might seek to add some hardware logic to the CPU which would look for, say, a particular rather unusual sequence of two instructions in a program. Should that sequence appear, the added logic might disable storage protection for just a few dozen microseconds. Such a small "hole" in the hardware would be quite sufficient for the user to (1) access anyone's file; (2) cause a system crash; (3) modify anyone's file.

User-controllable changes could be implemented in many ways, with many modes of control and action besides this example (which was, however, one of the more effective schemes contemplated). Countermeasures to such controllable changes will be considered below, along with ways in which a subverter might try to anticipate countermeasures.

Countermeasures to subversion

As implied earlier, anyone who has sufficient access to the CPU to install his own "design changes" in the hardware is likely to put in a controllable change, since a fixed change would be discovered by even a simple software test infrequently performed. A user-controllable change, on the other hand would not be discovered by tests outside the user's quantum, and would be hard to discover even within it, as will become obvious.

The automatic microprogrammed test previously discussed would have a low probability of discovering a user-controllable hardware change. Consider an attempt by a user to replace his log-in number with the log-in number of the person whose file he wants to steal. He must execute a MOVE CHARACTERS instruction of length 12 to do this, requiring only about 31 microseconds for the 360/50 CPU to perform. A microprogrammed test occurring at timer interrupts—once each 16 milliseconds—would have a low probability of discovering such a brief security breach. Increasing the test rate, though it raises the probability, raises the overhead correspondingly. A test occurring at 16 *micro*second intervals, for example, represents a 15 percent overhead.

A reasonable question is whether a software test might do a better job of spotting user-controllable hardware changes. One would approach this task by attempting to discover changes with tests inserted in user programs in an undetectable fashion. One typical method would do this by inserting invisible breakpoints into the user's instruction stream; when they were encountered during the user's quantum, a software test of storage protection and PSW bit 15 would be performed.

A software test of this type could be written, and as will be discussed, such a software test would be difficult for a subverter to circumvent. Nevertheless, the drawbacks of this software test are severe. Reentrant code is required so that the software test can know (1) the location of the instruction stream, and (2) that no instructions are hidden in data areas. Requiring reentrant programs would in turn require minor changes to the ADEPT–50 Jovial compiler and major changes to the F-level Assembler. A small microprogram change would even be required, so that software could sense the difference between a fetch-protect interrupt and an execute-protect interrupt. Changes would be required to the ADEPT–50 SERVIS, INTRUP, DEBUG, and SKED modules. Were such a software test implemented, run-time overhead would likely be rather high for frequent breakpoint-insertions, since each breakpoint inserted would require execution of 50 or more instructions at run time. Lastly, programmers might not always like having to write reentrant code.

These implementation problems, though described for one specific test approach, are likely to extend to any test which attempts to interrupt running user pro-

From the collection of the Computer History Museum (www.computerhistory.org)

grams via software, since any such test must isolate and augment every user's instruction stream.

Although the above does not demonstrate that detecting user controllable subversion hardware is impossible, it should indicate that by no means is such detection trivial. It is even harder than the above may suggest, for, as the next section indicates, while the subverter is busily building his controllable changes into the CPU he may add insult to injury by inactivating anti-failure and anti-subversion hardware.

Defeat of countermeasures

Hardware countermeasures that are easy to install are easy to remove. Redundant logic can be disabled with the same wire cutters or wire-unwrapper as other logic. This is also true of state-testing and parity-check logic. Alternatively, a clever subverter may add logic that picks up a test signal and uses it to inactivate his own logic while the test is performed. The subverter may, alternatively, arrange things so that tests always pass during his quantum.

Microprogrammed countermeasures are somewhat more difficult for the subverter to inactivate, since he must decode particular ROS addresses to know that a microprogrammed test routine has begun. Once he has done this, though, he may proceed as if he were dealing with hardware, since on a short-term basis, microprogram contents and addresses do not change. The net result of putting countermeasures in ROS rather than in logic is to raise the "work function" for subversion; that is, the subverter must do more work to achieve his goal, but he can reach it.

On the other hand, a thorough software test, such as the breakpoint test previously described, would be nearly impossible to defeat by hardware techniques. This is because of the transient nature of software: a subverter would have to decode program variables such as addresses via hardware; then even minor changes in the breakpoint-placing routines would make that hardware useless. One must recall, however, the large implementation and overhead problems inherent in a user-interrupting software test. In summary, countermeasures can be devised which have a high "work function," but they entail major costs in implementation and system efficiency.

Two assumptions have been inherent in this discussion; namely, that the subverter has both knowledge of system hardware (including subversion countermeasures) and means of changing the hardware. This need not be the case, but whether it is depends on administrative rather than technical considerations. Administrative considerations are the next subject.

Administrative policy

Special handling of hardware documentation and engineering changes may be worthwhile when commercial lines of computers are used for secure time-sharing. First, if hardware or microprograms have been added to the computer to test for failures and subversion attempts, the details of the tests should not be obtainable from the computer manufacturer's worldwide network of sales representatives. The fact that testing is done and the technical details of that testing would seem to be legitimate security objects, since a subverter can neutralize testing only if he knows of it. Classification of those documents which relate to testing is a policy question which should be considered. Likewise, redundant hardware, such as a second copy of the PSW bit 15 logic, might be included in the same category.

The second area is that of change control. Presumably the "Customer Engineer" (CE) personnel who perform engineering changes have clearances allowing them access to the hardware, but what about the technical documents which tell them what to do? A clever subverter could easily alter an engineering-change wire list to include his modifications, or could send spurious change documentation. A CE would then unwittingly install the subverter's "engineering change." Since it is asking too much to expect a CE to understand on a wire-by-wire basis each change he performs, some new step is necessary if one wants to be sure that engineering changes are made for technical reasons only. In other words, the computer manufacturer's engineering changes are security objects in the sense that their integrity must be guaranteed. Special paths of transmittal and post-installation verification by the manufacturer might be an adequate way to secure engineering changes; there are undoubtedly other ways. It is clear that a problem exists.

Finally, it should be noted that the 360/50 ROS storage elements, or any equivalent parts of another manufacturer's hardware that contain all system microprogramming, ought to be treated in a special manner, such as physically sealing them in place as part of hardware certification. New storage elements containing engineering changes are security objects of even higher order than regular engineering-change documents, and should be handled accordingly, from their manufacture through their installation.

GENERALIZATIONS AND CONCLUSIONS

Some general points about hardware design that relate to secure time-sharing and some short-range and long-range conclusions are the topics of this section.

From the collection of the Computer History Museum (www.computerhistory.org)

Fail-secure vs. fail-soft hardware

Television programs, novels, and motion pictures have made it well known that if something is "fail-safe," it doesn't blow up when it fails. In the same vein, designers of high-reliability computers coined the term "fail-soft" to describe a machine that degrades its performance when a failure occurs, instead of becoming completely useless. It is now proposed to add another term to this family: "Fail-secure: to protect secure information regardless of failure."

The ability to detect failures is a prerequisite for fail-secure operation. However, all system provisions for corrective action based on failure detection must be carefully designed, particularly when hardware failure correction is involved. Two cases were recently described wherein a conflict arose between hardware and software that had been included to circumvent failures.* Automatic correction hardware could likewise mask problems which should be brought to the attention of the System Security Officer via security software.

Clearly, something between the extremes of system crash and silent automatic correction should occur when hardware fails. Definition of what *does* happen upon failure of critical hardware should be a design requirement for fail-secure time-sharing systems. Fail-soft computers are not likely to be fail-secure computers, nor vice versa, unless software and hardware have been designed with both concepts in mind.

Failure detection by faulty system operation

Computer hardware logic can be grouped by the system operation or operations it helps perform. Some logic—for example, the clock distribution logic—helps perform only one system operation. Other logic—such as the read-only storage address logic in the 360/50—helps perform many system operations, from floating point multiplication to memory protection interrupt handling. When logic is needed by more than one system operation, it is cross-checked for proper performance: Should an element needed for system operations A and B fail, the failure of system operation B would indicate the malfunction of this portion of operation A's logic.

Such interdependence is quite useful in a fail-secure system, as it allows failures to be detected by faulty system operation—a seemingly inelegant error detection mechanism, yet one which requires neither software nor hardware overhead. Some ideas on its uses and limitations follow.

The result of a hardware logic failure can usually be defined in terms of what happens to the system operations associated with the dead hardware. Some logic failure modes are detectable, because they make logic elements downstream misperform unrelated system operations. Analysis will also reveal failure modes which spoil only the system operation which they help perform. These failures must be detected in some other way. There are also, but more rarely, cases where a hardware failure may lead to an operation failure that is not obvious. In the 360/50, a failure could cause skipping of a segment of a control microprogram that wasn't really needed on that cycle. Such failures are not detectable by faulty system operation at least part of the time.

Advantage may be taken of this failure-detection technique in certifying hardware to be fail-secure as well as in original hardware design. In general, the more interdependencies existing among chunks of logic, the more likely are failures to produce faulty system operation. For example, in many places in a computer one finds situations as sketched in Figure 1. Therein,

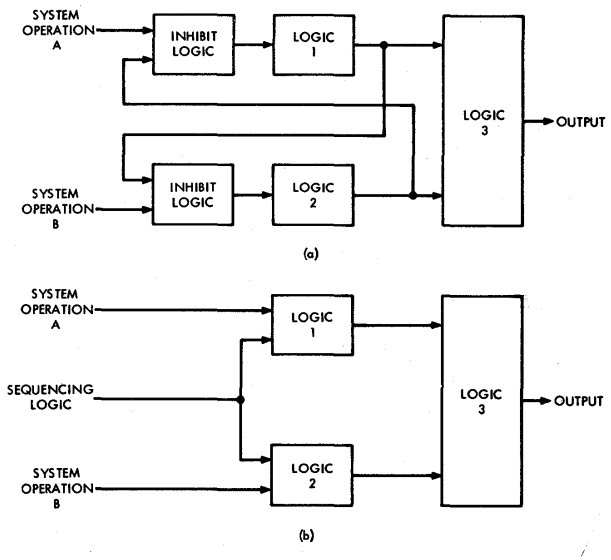

Figure 1—Inhibit logic vs sequencing logic

*At the "Workshop on Hardware–Software Interaction for System Reliability and Recovery in Fault-Tolerant Computers," held July 14–15, 1969 at Pacific Palisades, California, J. W. Herndon of Bell Telephone Labs reported that a problem had arisen in a developmental version of Bell's "Electronic Switching System." It seems that an elaborate setup of relays would begin reconfiguring a bad communications channel at the same time that software in ESS was trying to find out what was wrong. R. F. Thomas, Jr. of the Los Alamos Scientific Laboratory, having had a similar problem with a self-checking data acquisition system, agreed with Herndon that hardware is not clever enough to know what to do about system failures; software failure correction approaches are preferable.

From the collection of the Computer History Museum (www.computerhistory.org)

TABLE 1—Control Signal Error Detection by Odd Parity Check on Odd-Length Data Field

DATA BITS *012*	*P*	*MEANING*
000	0	data error or control logic error*
000	1	0
001	0	1
001	1	data error
010	0	2
010	1	data error
011	0	data error
011	1	3
100	0	4
100	1	data error
101	0	data error
101	1	5
110	0	data error
110	1	6
111	0	7
111	1	data error or control logic error**

*Control logic incorrectly set all bits to zero.
**Control logic incorrectly set all bits to one.

System Operation A needs the services of Logic Group 1 and Logic Group 3, while System Operation B needs Logic Group 2 and Logic Group 3. Note at this point that, as above, if System Operation A doesn't work because of a failure in Logic Group 3, we have concurrently detected a failure in the logic supporting System Operation B.

A further point is made in Figure 1. Often System Operations A and B must be mutually exclusive; hardware must be added to prevent simultaneous activation of A and B. Two basic design approaches may be taken to solve this problem. An "inhibiting" scheme may be used, wherein logic is added that inhibits Logic Group 1 when Logic Group 2 is active, and vice versa. This approach is illustrated by Figure 1(a). Alternatively, a "sequencing" scheme may be used, wherein logic not directly involved with 1 or 2—such as system clock, mode selection logic, or a status register—defines when A and B are to be active. This approach is illustrated by Figure 1(b).

Now, "inhibit" logic belongs to a particular System Operation, for its function is to asynchronously, on demand, condition the hardware to perform that System Operation. It depends on nothing else; if it fails by going permanently inactive, only its System Operation is affected, and no alarm is given. On the other hand, "sequencing" logic feeds many areas of the machine; its failure is highly likely to be detected by faulty system operation.

A further point can be made here which may be somewhat controversial: that an overabundance of "inhibit"-type asynchronous logic is a good indicator of sloppy design or bad design coordination. While a certain amount must exist to deal with asynchronous pieces of hardware, often it is put in to "patch" problems that no one realized were there till system checkout time. Evidence of such design may suggest more thorough scrutiny is desirable.

System Operations can be grouped by their frequency of occurrence: some operations are needed every CPU cycle, some when the programmer requests them, some only during maintenance, and so on. Thus, some logic which appears to provide a cross-check on other logic may not do so frequently or predictably enough to satisfy certification requirements.

To sum up, the fact that a system crashes when a hardware failure occurs, rather than "failing soft" by continuing to run without the dead hardware, may be a blessing in disguise. If fail-soft operation encompasses hardware that is needed for continued security, such as the memory protection hardware, fail-soft operation is not fail-secure.

Data checking and control signal errors

Control signals which direct data transfers will often be checked by logic that was put in only to verify data purity. The nature and extent of this checking is dependent on the error-detection code used and upon the length of the data field (excluding check bits).

What happens is that if logic fails which controls a data path and its check bits, the data will be forced to either all zeros or all ones. If one or both of these cases is illegal, the control logic error will be detected when the data is checked. (Extensive parity checking on the 360/50 CPU results in much control logic failure detection capability therein.) Table 1 demonstrates an example of this effect; Table 2 describes the conditions for which it exists for the common parity check.

TABLE 2—Control Signal Error Detection by Parity Checking

DATA FIELD LENGTH:	PARITY:	CONTROL LOGIC ERROR CAUSES: all zeros	all ones
even	odd	CAUGHT	MISSED
even	even	MISSED	CAUGHT
odd	odd	CAUGHT	CAUGHT
odd	even	MISSED	MISSED

From the collection of the Computer History Museum (www.computerhistory.org)

CONCLUSIONS

From a short-range viewpoint, 360/50 CPU hardware has some weak spots in it but no holes, as far as secure time-sharing is concerned. Furthermore, the weak spots can be reinforced with little expense. Several alternatives in this regard have been described.

From a longer-range viewpoint, anyone who contemplates specifying a requirement for hardware certification should know what such an effort involves. As reference, some notes are appropriate as to what it took to examine the 360/50 memory protection system to the level required for meaningful hardware certification. The writer first obtained several publications which describe the system. Having read these, the writer obtained the logic diagrams, went to the beginning points of several operations, and traced logic forward. Signals entering a point were traced backward until logic was found which would definitely cause faulty machine operation outside the protection system if it failed. During this tedious process, discrepancies arose between what had been read and what the logic diagrams appeared to show. Some discrepancies were resolved by further study; some were accounted for by special features on the SDC 360/50; some remain.

After logic tracing, the entire protection system was sketched out on eight $8\frac{1}{2} \times 11$ pages. This drawing proved to be extremely valuable for improving the writer's understanding, and enabled failure-mode charting that would have been intractable by manual means from the manufacturer's logic diagrams.

For certifying hardware, documentation quality and currentness is certainly a problem. The manufacturer's publications alone are necessary but definitely not sufficient, because of version differences, errors, oversimplifications, and insufficient detail. Both these and machine logic diagrams are needed.

Though the hardware certification outlook is bleak, an alternative does exist: testing. As previously described, it is possible to require inclusion of low-overhead functional testing of critical hardware in a secure computing system. The testing techniques, whether embedded in hardware, microprograms, or software, could be put under security control if some protection against hardware subversion is desired. Furthermore, administrative security control procedures should extend to "Customer Engineer" activity and to engineering change documentation to the extent necessary to insure that hardware changes are made for technical reasons only.

Careful control of access to computer-based information is, and ought to be, of general concern today. Access controls in a secure time-sharing system such as ADEPT–50 are based on hardware features.[7] The latter deserve scrutiny.

REFERENCES

1 L MOLHO
Hardware reliability study
SDC N-(L)-24276/126/00 December 1969

2 R LINDE C WEISSMAN C FOX
The ADEPT-50 time-sharing system
Proceedings of the Fall Joint Computer Conference Vol 35 p 39–50 1969
Also issued as SDC document SP-3344

3 W H WARE
Security and privacy in computer systems
Proceedings of the Spring Joint Computer Conference Vol 30 p 279–282 1967

4 W H WARE
Security and privacy: Similarities and differences
Proceedings of the Spring Joint Computer Conference Vol 30 p 287–290 1967

5 S G TUCKER
Microprogram control for system/360
IBM Systems Journal Vol 6 No 4 p 222–241 1967

6 G C VANDLING D E WALDECKER
The microprogram control technique for digital logic design
Computer Design Vol 8 No 8 p 44–51 August 1969

7 C WEISSMAN
Security controls in the ADEPT-50 time-sharing system
Proceedings of the Fall Joint Computer Conference Vol 35 p 119–133 1969
Also issued as SDC document SP-3342

From the collection of the Computer History Museum (www.computerhistory.org)

From the collection of the Computer History Museum (www.computerhistory.org)