AARDVARK:

A HIGHLY VIRUS–RESISTANT COMPUTER ARCHITECTURE

A Thesis

Submitted to the Faculty

of

Rose–Hulman Institute of Technology

by

Charles James Wilson

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical Engineering

November 1993

March 1999

Second Printing, with revisions

Printed in the United States of America

ABSTRACT

Wilson, Charles James, Rose–Hulman Institute of Technology, November 1993. Aardvark: A Highly Virus-Resistant Computer Architecture. Major Professor: Bruce A. Black.

Aardvark is a highly virus–resistant computer architecture based on existing technologies. The virus–resistant nature stems from the use complementary mechanisms: dual level system routines in ROM (split SROM), physically separate instruction and data spaces (separate I and D) and executable image encryption. These three mechanisms protect the computer system from low level operating system routines that circumvent high level security schemes by accessing the computer system's drivers directly, modification of in–memory executables and modification of non–loaded executable images respectively.

Dedicated to the people who believed more in me than I did in myself, especially my father whose encouragement meant more than I could have ever possibly expressed.

ACKNOWLEDGEMENTS

You do not just wake up one morning and decide, “I feel brilliant today; I think I’ll write a master's thesis on virus–resistant computer architectures.” My father encouraged me even when I was destroying things around the house to realize my designs. Darrel Criss and Mike Atkins challenged my mind during my undergraduate studies. Wilford Stratton, Donald Morin and Alfred Schmidt listened to and encouraged my ideas for enhancing man’s ability to utilize technology. Emmett Black taught me how to deal with corporate types at General Electric Space Systems. David Wise showed me what it meant to be a debugging expert. Frank Young thought a thesis on computer virus protection would be a pretty neat idea. Philip Fowler believed that I could do it.

Charles James Wilson

Rose–Hulman Institute of Technology

November 1993

PREFACE

Because of the complex nature of computer viruses and the complex way in which they interact with computer systems, the solution is not simply explained. Many times in this document it is necessary to refer to elements of Aardvark which have not yet been introduced. Also, since Aardvark by necessity deals with a software and operating system issue with a hardware–based solution, the reader is assumed to be comfortable with this kind of approach. The ideal reader is one who has a knowledge of computer virus pathology, is highly conversant with computer architecture and has a firm grasp of computer system software design. Realizing that this is not a likely combination of knowledge bases, I have attempted to explain those concepts which may be unfamiliar to some readers. In order to aid the reader further, there is an extensive section of references separated into groups following the main body of the text.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

Aardvark - A pseudo–acronym for A Highly Virus–resistant Computer Architecture (AHVRCA).

appending - A method of viral infection in which viral code is appended to the end of an application’s exit sequence code.

Cricket - The checksumming logic within an Aardvark–based computer.

Data Encryption Standard - An NSA supported standard for the encryption of computer data.

DES (see Data Encryption Standard)

designer virus - A type of virus tailored to attack a specific application.

Dolphin - The decryption logic within an Aardvark–based computer.

Gopher - The NVRAM lookup logic within an Aardvark–based computer.

Kinkajou - The Touch Memory™ interface logic within an Aardvark–based computer.

National Security Agency (NSA)- An agency of the federal government responsible for the communications security of the United States and is highly concerned with the use of cryptologic techniques as used in communication.

NSA (see National Security Agency)

NVRAM - Non–Volatile Random Access Memory. Memory which is not subject to loss of data when external power is removed. This is accomplished through the use of an internal battery and circuitry which detects the loss of external power.

NVSRAM - Non–Volatile Static Random Access Memory. NVRAM which is in the static memory class.

overwriting - A method of viral infection accomplished by completely overwriting the existing code segments with no attempt made to preserve the original application functionality.

PCMCIA (see Personal Computer Memory Card International Association)

Personal Computer Memory Card International Association - (alternately: People Can't Manage Creating Intelligible Acronyms). A 300 member consortium of computer manufacturers acting as a standards body and trade organization to develop and promote small form–factor personal computer devices. PCMCIA may be used to refer to a device interface or the device itself (c.f. SCSI).

prepending - A method of viral infection in which virus code is prepended to the beginning of the execution sequence of an application.

routine replacement - A method of viral infection in which the virus replaces the application's routines with its own.

shell - A method of viral infection in which viral code is both prepended and appended to the application’s code.

SROM - System ROM. System routines stored in Read-Only Memory used to store large collections of common subroutines. On IBM PC–based systems the SROM is referred to as the BIOS (Basic Input/Output System).

tail-patching - A method of viral infection in which viral code is appended to the end of an application’s code.

Touch Memory™ - A trademarked serial data stream delivery system developed by Dallas Semiconductor for use in hostile environments.

Trojan Horse - A hostile application disguised as a harmless one.

Worm - A self–contained application, usually found within computer networks, which has the operational features of a computer virus but does not infect other applications in its replication process.

1. INTRODUCTION

1.1. Statement of the Problem

Since the advent of the computer, individuals have created programs designed to gain access to data to which they would not normally have access to and possibly damage or destroy this data. The reasons for this type of activity range from curiosity to revenge to espionage.

1.2. Importance of the Problem

Any loss of data or resource control (e.g., when a computer system is overwhelmed by a parasitic program that has instantiated itself and is consuming most or all of the processing capabilities) is almost always costly to the owner of a computer system. This cost includes time lost to removing a detected virus from the system and the associated scanning of tape backups to assure their integrity, time lost in data restoration or in some cases reconstruction, processing time lost to virus detection software and time and money needed to restore or replace faulty software. Any advances in the area of computer system design which address this area would produce considerable savings.

1.3. Purpose of Aardvark

Aardvark is a computer architecture designed to be highly virus–resistant. It is intended to protect the integrity of application executables throughout their lifetime, from installation, through execution(s) until their final deinstallation. Since the main mechanism of viral propagation is through the instantiation of viral code into other application executables, this effectively prevents the spread of viruses within the computer system.

Additionally, the architecture provides for a dual level interface to the system routines in ROM (SROM). The interface is set up in such a way as to prevent applications currently in execution from directly accessing low level driver routines. This protects the computer system from attacks which use these low level routines to do damage.

Upon detection of an attempted viral attack, the computer system informs the user that such an attack is taking place or has taken place. Further, the computer system instructs the user as to what action may be taken to eliminate the threat posed.

The design of Aardvark also allows for higher confidence in data files created by 'trusted' applications. Data files created by these trusted applications may only be modified by their creator. Other applications would be allowed read only access to the files.

1.4. What Aardvark Will Not Do

Aardvark is designed to deal with the spread of viruses to other applications. It does not prevent a virus from being introduced into the computer system, but rather prevents any virus which has been introduced through an non–trusted application from spreading. If a virus is introduced into an Aardvark–based computer system there is still the possibility that it may damage non–trusted application executables or data files.

2. BACKGROUND MATERIAL

2.1. Summary Statement

To properly evaluate the solution proposed in this paper, it is necessary to understand the techniques employed by virus programs.

2.2. Nomenclature

2.2.1. Overview

Computer viruses have evolved from simple programs which were crude in both their method of attack and their internal design into sophisticated programs capable of performing highly intricate operations on the file system and user files within a computer system. Some elude virus scanners and may even infect applications being used to detect and eliminate viruses. It is important to establish some parameters with which to bound what will be considered a computer virus within the scope of this work.

2.2.2. Trusted and Non–trusted Applications

Throughout this work there will be constant references to ‘trusted’ and ‘non–trusted’ applications. Within the scope of this work a trusted application shall be one whose application executable has been verified to contain no viruses and has been protected using the mechanisms provided by Aardvark. A non-trusted application is one that does not have both of these attributes.

2.2.3. Propagation

One of the most prominent features of a computer virus is propagation. The faster a virus spreads into other applications the more quickly it tends to be detected. Some viruses target specific application executables, such as system applications, so that they may spread quickly. Others wait until an infected application executable is moved onto another computer system.

2.2.4. Pathogenesis

A computer virus may be pathogenic in nature. Some viruses merely announce that they are present within an application in order to bring attention to the author of the virus. Others are designed to destroy data or applications. Unfortunately, viruses designed to attack a computer’s file system may inadvertently damage data structures which were not the target of the viral attack.

2.2.5. Carriers

Some computer viruses make use of carriers in their propagation. These carrier applications are infected with the virus, but the virus will not become active or be spread further until placed into a suitable environment.

2.2.6. Trojan Horses

A non-virus which tends to be grouped with viruses is the Trojan horse, an application which contains code which does not perform as the user is led to believe it will. These type of applications are usually advertised as system utilities which when invoked actually cause damage to the computer system rather than improve its performance. Trojan horses are not within the scope of this work.

2.2.7. Worms

A worm is a self-contained, virus-like application. Unlike a virus, no applications are infected when a worm replicates. It may be pathogenic in nature. The usual home for a worm is within a networked computer environment where replication is followed by a search for additional hosts on the network. The most famous worm in recent history is the Internet Worm which crippled many installations on the Internet because of its unusually rapid replication rate [1,2,3,4,5,6,7,8]. This type of application is also not within the scope of this work.

2.3. Targets of Attack

2.3.1. File System

2.3.1.1. Overview

The first and probably the most complex type of attack on a computer system is one against the computer's file system. This type of attack requires intricate knowledge of the operating system if the intent is to infect the system without rendering the system completely inoperable. The ever changing nature of computer file system implementations along with the idiosyncrasies of individual computer systems make it unlikely that a virus which attacks the computer’s file system will be written in such as way as to not cause unintended damage. There are two broad categories of computer file system attack: nuisance and malicious.

2.3.1.2. Nuisance Incursion

The nuisance attack consists of the modification of the computer system’s file structures. This modification may include alteration of directory structures (e.g., creation/access/modification dates or iconographic attachments). The damage from this type of attack is for the most part inconsequential. Because this type of attack is intended to be subtle and non-malicious, the damage incurred may take a great deal of time to discover.

2.3.1.3. Malicious Incursion

The other and more prevalent type of attack on the computer file system is that of malicious incursion. This type of attack is designed to cause direct damage to the data structures which allow the computer system to function. The targets of malicious incursion include file size, file segmentation links, extension, access permissions, etc. The damage caused by malicious incursion is often severe. The lost or altered data cannot be easily recovered or repaired. The usual recourse is to treat the entire computer file system as corrupt and reload the computer system from distribution media and pre-incursion backups.

2.3.2. Data Files

A second possible target for a virus is data files. This attack may take several forms. The data contained within a data file may be corrupted (i.e., garbage data may be written onto the valid data). The data may be encrypted. It may be reordered in such a way that renders it unusable. The data may be hidden elsewhere in the computer system storage system.

Each of these forms of attack results in the loss of access to possibly valuable data. Database and spreadsheet data files are especially sensitive to changes in their content.

The data encryption attack has been used as a form of extortion [19]. The virus in this case informed the user that their files had been encrypted and that they would be able to retrieve the files if they sent a specified amount of money to a post office box. Unfortunately for the writers of viruses, it is very easy to trace such transactions.

Although the protection of data files is not the main focus of the Aardvark computer architecture, an architectural extension to Aardvark is described in the section on data file security (see Data File Security page 66).

2.3.3. Application Executables

The final type of attack is against the application executables resident on the computer systems. It is this type of attack that is primarily responsible for the propagation of computer viruses. The protection of the application executable is the primary anti–virus service which the Aardvark computer architecture provides. This type of attack will be discussed in detail in the following section.

2.4. Application Executable Infection

2.4.1. Overview

In order to explain the methods viruses use to infect application executables, it is necessary to introduce a model of an application executable. In order to reduce the complexity of discussion, a simplified application executable model will be used within this paper.

In its most simplified form, an application executable is merely a single stream of instructions whose execution begins at the start of the application executable. This type of application has, however, become a very rare creature. The application executable model which will be used throughout this thesis is seen in Figure 1.

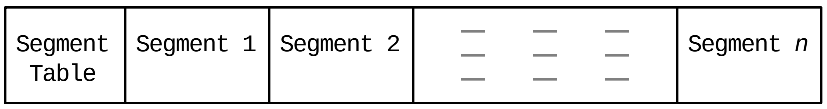


Figure 1 - Application Executable Model

The application executable is composed of a segment table and a series of code segments. The segment table is a data structure made up of a set of pointers to the various code segments (see Figure 2).

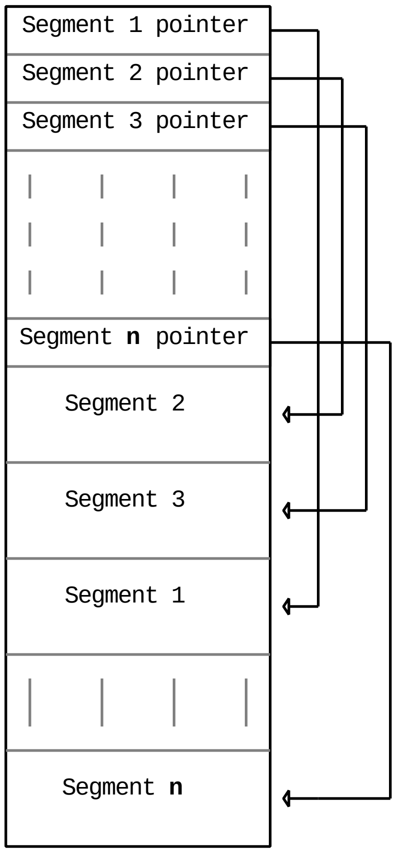


Figure 2 - Pointer/Segment Relationship

Notice that the segments are not necessarily stored sequentially. There is no guarantee that the linker used to generate the application executable will do so.

2.4.2. Methods of Viral Infection

Three primary methods are used by viruses to instantiate a virus into an application executable. These methods include: tail patching, overwriting and routine replacement (see Figures 3, 4 and 5). Other methods of instantiation are variations of these [16].

2.4.3. Tail Patching

Tail patching an application executable involves extending the segment pointer table and addition of new segments (see Figure 3 on page 12). The technique is called tail patching because the viral code is added to the end of the application executable to prevent modification of internal references created by the compiler and linker. The viral code segment is added to the end of the application executable, possibly extending its length. The segment table is extended to allow the inclusion of an additional entry for the viral code segment. The viral code segment is then made the entry point for the application executable. In order to maintain the appearance of normal functionality, the last operation of the viral code segment is a call to the updated segment 1 pointer. This type of infection allows the application to function normally.

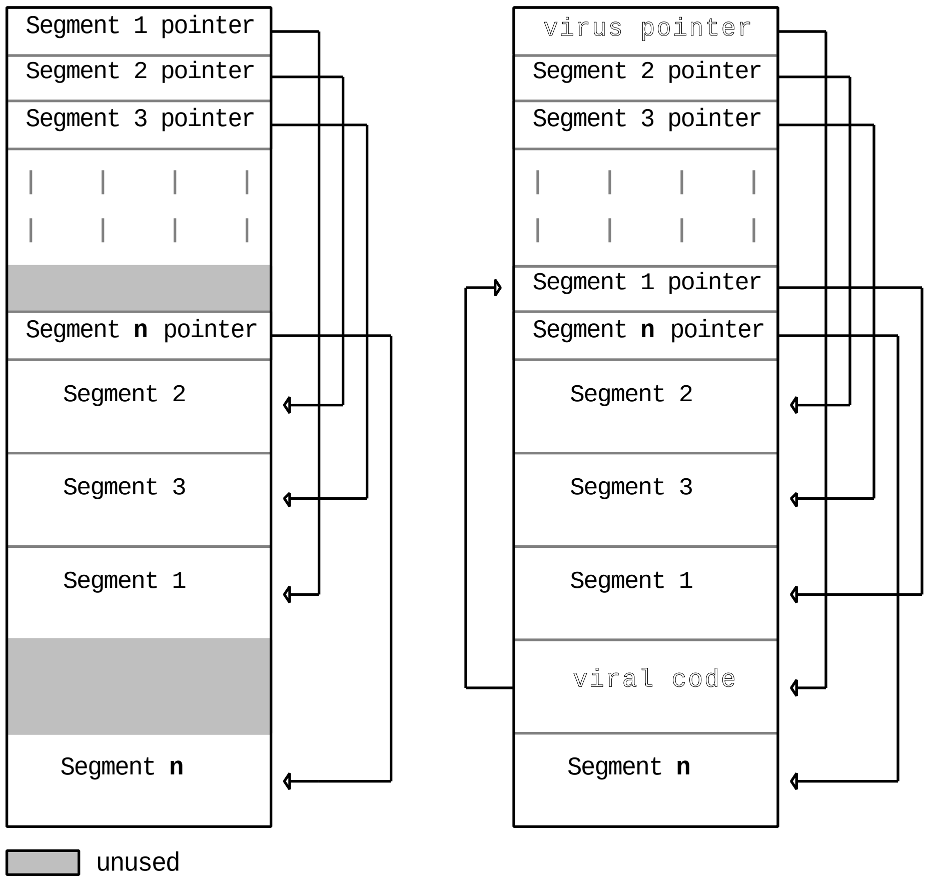


Figure 3 - Tail Patching Infection

A tail patch infection is very difficult to detect unless the viral code adds substantial overhead to the application usually causing a long apparent load time. Stealth is the hallmark of this method since many applications may become infected before the virus is detected.

In some cases, the virus creator attempts to place the viral code in the exit path of the application instead of the entry path. This can present a problem since there is no guarantee that any given application executable will have a single exit path. This type of attack is also know as prepending or appending since the viral code is prepended of appended to the operational sequence of the application.

A combination of prepending and appending may also be used. This yields what may be referred to as a shell. The application is allowed to operate normally, but both its instantiation and termination code are bracketed by viral code. As mentioned above, problems arise when the application executable has multiple exit paths. Should a virus infect an application executable with multiple exit paths, it is likely that the application will perform in an erratic manner since the viral termination code would not always execute.

2.4.4. Overwriting

The most simplistic method of viral infection is to simply overwrite an application executable (see Figure 4 on page 14). The initial application segment pointer is replaced by the viral code pointer and the initial segment is replaced by the viral code segments. No attempt is made to preserve the infected application executable.

With this method, the size of the application executable remains unchanged. This causes virus checking by size difference to fail. On the down side, infection is quite obvious when an application executes.

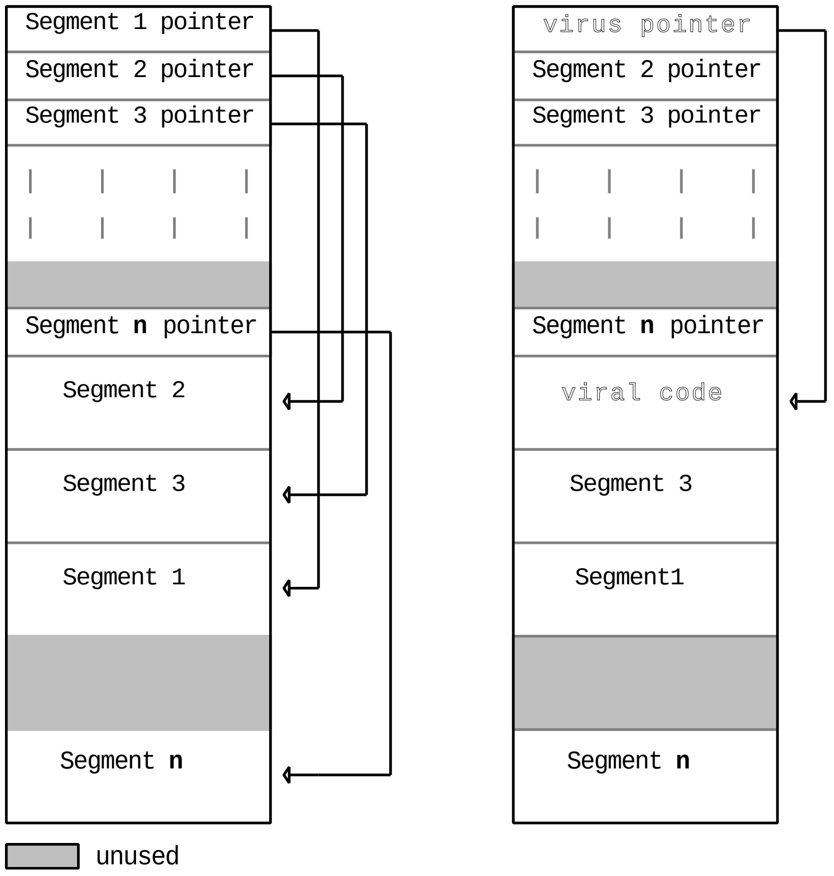


Figure 4 - Overwriting Infection

2.4.5. Routine Replacement

Routine replacement is a sophisticated form of overwriting. Instead of overwriting the initial segment, an infrequently used segment is replaced by the viral code (see Figure 5 on page 15). The application being infected is examined to identify an infrequently used routine. Alternately infection may be limited to a particular, well understood application. The viral code will have an effective detection delay established for it. The application may be able to function for an extended period before the damage to it is detected.

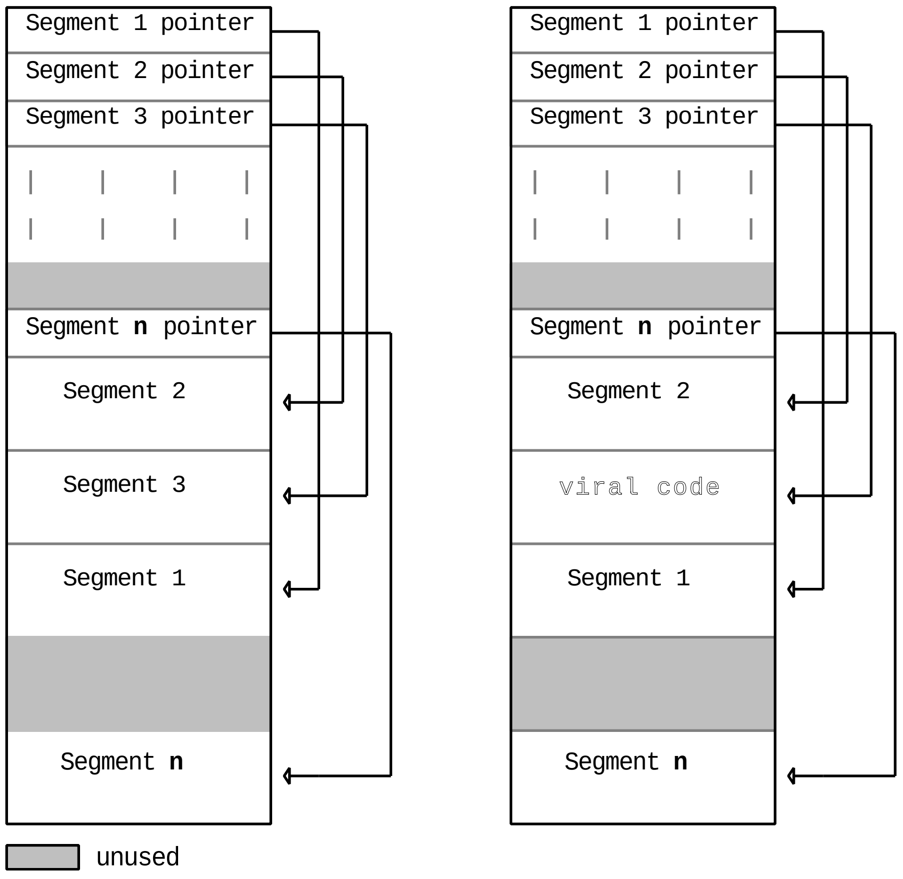


Figure 5 - Routine Replacement Infection

2.5. Virus Self-protection

One of the latest advances in the area of virus creation has been stealth. The first viruses were extremely obvious. Soon after infection, the user would be well aware of their presence. This is no longer the case. Numerous mechanisms of infection delay are used in the current generation of viruses. These range from simple reference to the system clock to sophisticated monitoring of system resource and application usage.

If a virus is detected quickly, it is eliminated just as quickly. If, however, a virus does not make its presence known for a period of time, there is a much greater likelihood that the virus will be able to spread to additional media and/or computer systems.

The latest generation of viruses comes equipped with anti-viral application detection and bypassing, as well as viral code encryption to protect themselves. These techniques greatly hamper attempts to maintain a computer system free from unwanted viral code [17].

The specifics of these protection techniques are not within the scope of this paper. Detailed explanations of the various virus self-protection techniques can be found in David Ferbrache’s book, <u>A Pathology of Computer Viruses</u> [15].

2.6. Approaches to Virus Protection

2.6.1. Traditional Approaches

2.6.1.1. Boot-time Scanning

The most time-honored method of dealing with the threat of computer viruses is boot-time scanning. This requires a program that runs each time the computer is booted. Once in execution, this virus checker scans the system for any signs of viruses or viral activity. The amount of functionality varies greatly from applications that merely report the presence of a virus to those that can correct any problems encountered.

There are always tradeoffs between speed, size and functionality. Where one application may always be present in the computer's memory, another may run once and exit. The former would by necessity, on most microcomputers, be small to satisfy the requirements of background tasks. The latter need not deal with this constraint.

The problem with this technique of virus detection is that it assumes that the virus detection application has not itself become the victim of a viral attack. This would render the virus checking application inactive and leave the user with the erroneous impression that the system is secure. This concern is addressed in the section on application self-checking (see <u>Application Self-checking</u> page 18).

2.6.1.2. Application Executable Checksumming

Currently the most popular idea being discussed in the virus news group on the Internet is application protection by checksumming. This technique requires that upon installation a checksum of the application executable be made and entered into a database. Periodically, the checksum is verified to determine if the applications currently logged have been modified. If so, the user can be informed that the system has been attacked and that it is necessary to perform an analysis of the system using a virus scanner and also reload any affected applications.

The weakness of this methodology is similar to that of application self-checking. The database and application that perform the verification of the application checksums are resident in the memory system of the computer. If a virus is aware of the presence of such an application, it may infect registered applications and then modify the corresponding entries in the checksum database.

2.6.1.3. Application Self–checking

An anti–virus technique that is gaining popularity among software publishers is self–checking. The application runs code which looks at multiple elements of the application executable for congruity to a set of established data. If the self–check fails, the application reports that it has been tampered. For obvious reasons, virus detection software usually employs this type of self–protection.

Applications protected by this technique are more difficult to attack. There is concern, however, that these applications may become the target of ‘designer viruses’. These viruses would be written to attack specific applications. Creation of this type of virus would require that the application first be disassembled to determine what, if any, self–checking code is in place. It is a simple task to infect the application and then either modify the baseline data or eliminate the self–check code. The former would be simpler since the self–check code may be threaded through by other portions of the application to ensure that it is not tampered with.

2.6.2. Problems With Traditional Approaches

The problem inherent in all the traditional methods for virus protection is that they depend on the virus scanning or self–checking code to remain itself uncorrupted and capable of detecting viruses or alterations in the system state. Unfortunately, they respond very poorly when new types of viruses or exotic strains of existing viruses are introduced into the system.

2.6.3. Aardvark’s Approach

Aardvark is designed to assure the integrity of applications regardless of whether or not viruses are present on the system. The approach taken by Aardvark is to assure the integrity of the application executable by having an encrypted executable. This encrypted application executable is only decrypted on a segment-by-segment basis as it is loaded into instruction memory. This memory is available for reading only by the processor during the actual execution of instructions. An additional layer of protection is provided by logic which computes a checksum of the application executable prior to loading. This ensures that there have been no gross level modifications attempted. The SROM is protected from abuse through logic which validates low–level or hazardous system calls. This methodology effectively isolates any computer virus which attacks an Aardvark–based computer system.

2.7. Difficulties in Research

One might imagine that the subject of computer viruses would be one like any other. In order to study the subject, one should be able to go to a library, locate the catalog entries concerned with computer viruses, both book and periodical, and retrieve the pertinent information. Following the initial research, one would set up several machines and infect them with viruses acquired from various sources. The effects of the viruses on the systems could then be evaluated and possible solutions tested for effectiveness.

The reality of the situation is that doing any research in the area of computer viruses is very difficult. The creators of computer viruses wish to have their names famous but not their identities. The companies who design virus detection software do not wish to divulge the techniques used by virus writers to prevent would–be virus writers from learning them. For the same reason, sources to decompiled viruses cannot be found in any of the source code archives on the Internet.

Trying to set up a testing facility for virus testing is like trying to set up a bubonic plague test center. Everyone acknowledges the value of the work and good intentions and appreciate that plans to take every possible safety precaution, but they would prefer that you take your “toxic waste dump” to another state, thank you very much.

One of the greatest roadblocks in virus research is the parent of the Computer Security Center, the National Security

Agency (NSA). According to a public affairs officer with the NSA, “[Computer viruses are] just one of those things we don’t talk about [9,10].” Fred Cohen of the University of Cincinnati was told by an NSA employee that “You’re not going to do any research on viruses if we can help it …” It has been speculated that the NSA injected a computer virus into the Iraqi defense system during the Gulf War. The NSA has refused to comment on the story even though it is considered to be a hoax [5,11].

3. COMPUTER ARCHITECTURES

3.1. Conventional Computer Architecture

Many different strategies have been used to design computer architectures. In the past twenty years, the industry has settled on one particular architecture for microcomputer design. In this model, there is an SROM (System ROM) or kernel, a contiguous main memory, a host of internal I/O devices (e.g., keyboard, screen, mouse, fixed and removable media) and external I/O devices (e.g., printer ports, serial communications ports, network ports). All of these elements of the architecture provide it with a great deal of flexibility and configurability in that a stock machine may be ‘stripped down’ to allow for special purpose applications or enhanced as newer technologies present themselves. The portion of the traditional computer architecture I will focus on is the CPU to memory subsystem. The conventional microcomputer architecture’s CPU to memory subsystem (see Figure 6 on page 24) is very straightforward in both design and implementation. All major operating systems available today treat the entire body of memory as free for the application’s use. No distinction is made between what is code, what is application stack space and what is data. A program is unable to determine whether the code being called is in RAM or ROM. This is not to say that applications have free run of the entire address space of the machine because, in general, this is simply not the case. It merely points out that checks must be made to insure that applications do not overrun themselves or other applications including the operating system during the course of execution. It is while resident in memory that data is vulnerable. Applications make the assumption that data structures maintained by them are valid. If a second application were to corrupt these data structures, the first would be unable to detect any such corruption. Also, in order to execute an application, it must be present in memory. While in this state, checks must be made to ensure that an application does not read another’s code. This must be done in order to preserve the security of certain applications.

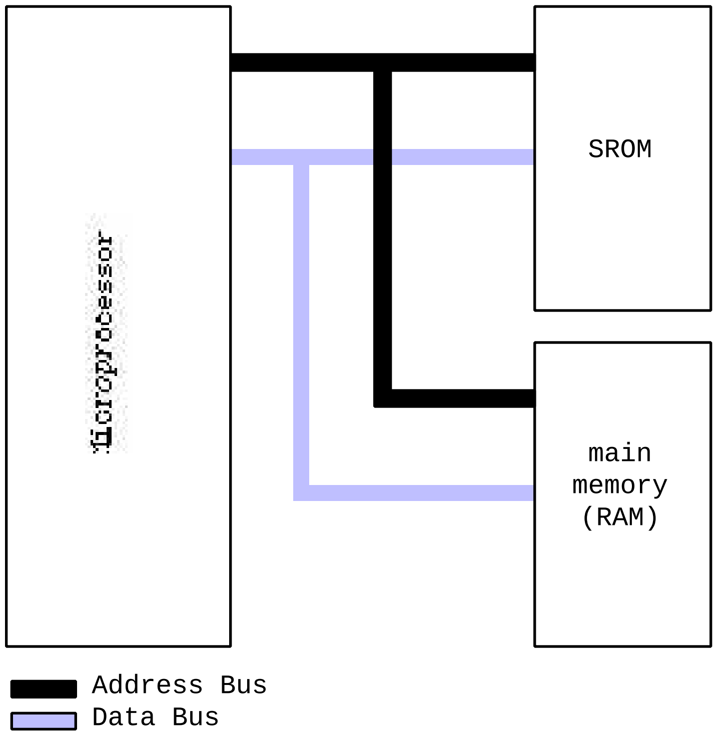


Figure 6 - Conventional Computer Architecture

3.2. Aardvark’s Computer Architecture

3.2.1. Description of Aardvark

Aardvark’s computer architecture is pictured in Figure 7 on page 25. The term virus–resistant is used as opposed to virus–proof because I do not believe that it is possible to have a completely virus–proof computer system so long as the elements of the operating environment (OS, user interface, etc.) exist in software and not exclusively in firmware and hardware. Although complete virus protection may be feasible in dedicated systems such as embedded microcontrollers, the continual upgrading required by most computer systems make this implementation technique highly unlikely.

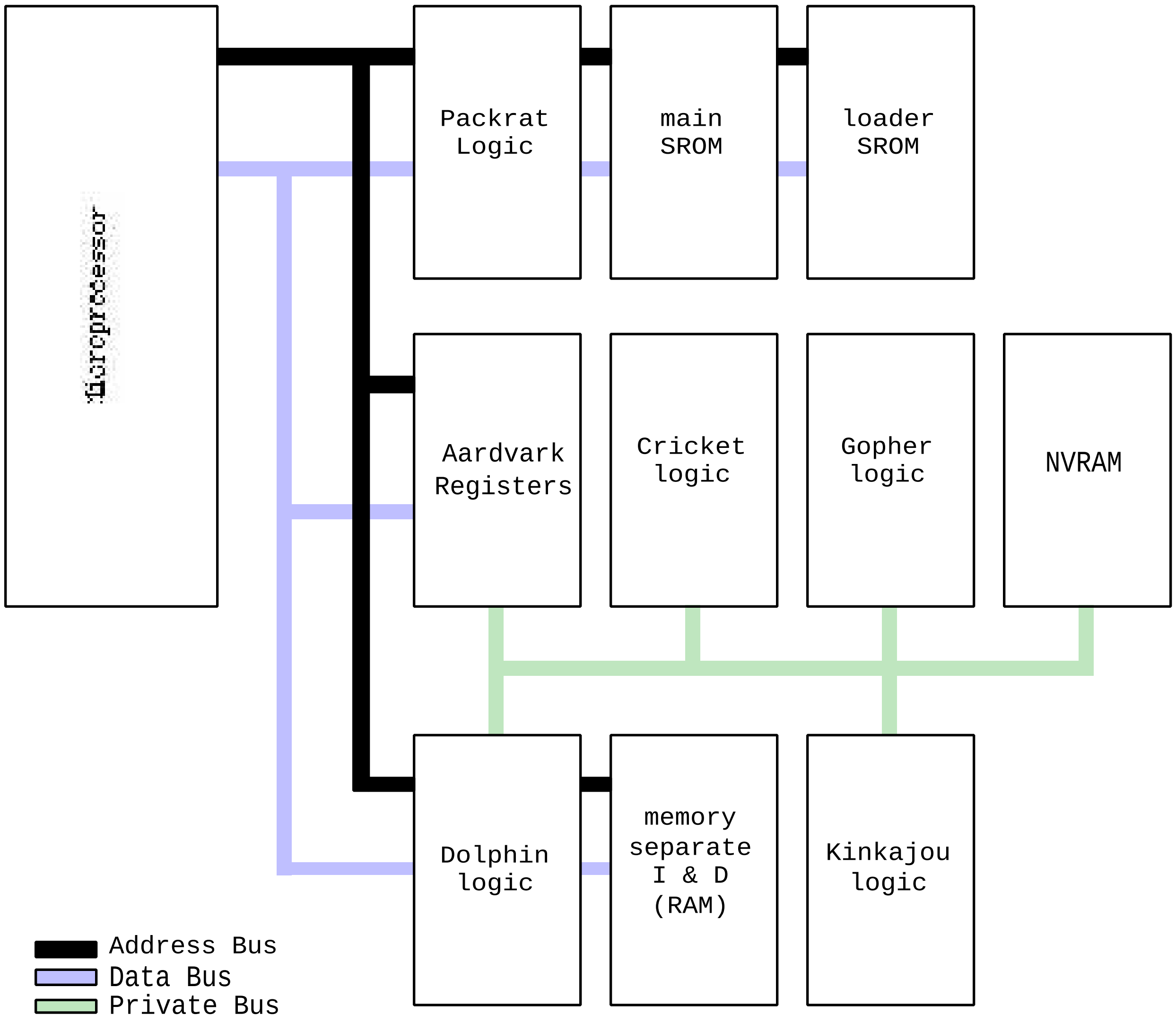


Figure 7 - Aardvark’s Computer Architecture

The chief technique used by Aardvark in ensuring the integrity of the environment for applications is the use of application encryption. Every ‘trusted’ application is stored in encrypted form. Only when the application executable is loaded into the computer system’s active memory is it decrypted. This decryption is performed within hardware. For additional security, the encryption is performed on the segment level. For purposes of this work, it will be sufficient to refer to application executables using a simplified segment model (see Figure 1 on page 9). The actual mechanism used to accomplish application loading and decryption are described in the section on application executable loading (see <u>Application Execution</u> page 45).

3.2.2. Description of Subsystems

Aardvark can be viewed as a main processor connected to a set of subsystems (see Figure 7 page 25). These subsystems interact to provide virus protection. Each subsystem is named for an animal having the attributes of the function performed by the subsystem. Briefly, the subsystems perform the following functions:

Cricket	gross level application checksumming
Dolphin	application executable segment decryption
Gopher	NVRAM access control
Kinkajou	Touch-Memory™ input control
Packrat	SROM access control

4. ARCHITECTURAL ELEMENTS

4.1. Introduction

The intent of Aardvark is to prevent viruses from spreading within a computer system. This section focuses on the architectural elements used to accomplish this. It is not possible to explain these elements in isolation in any satisfactory manner. I have attempted to present them in a way which provides the best basis for understanding the overall operational mechanisms used within Aardvark. At times there will be heavy use of forward references to material in related sections of the architecture.

In order to protect the integrity of trusted applications, it is necessary to maintain a set of baseline data by which the integrity of the application may be determined. This data is protected by physically separating it from direct access by the computer system through the use of fire walls in the form of highly autonomous subsystem interfaces and related control logic.

4.2. Main Processor/Subsystem Communication

The main processor and subsystems communicate via a set of dual–ported bidirectional off–processor registers (see Figure 8 on page 28). These registers act as buffers between the main processor and subsystems.

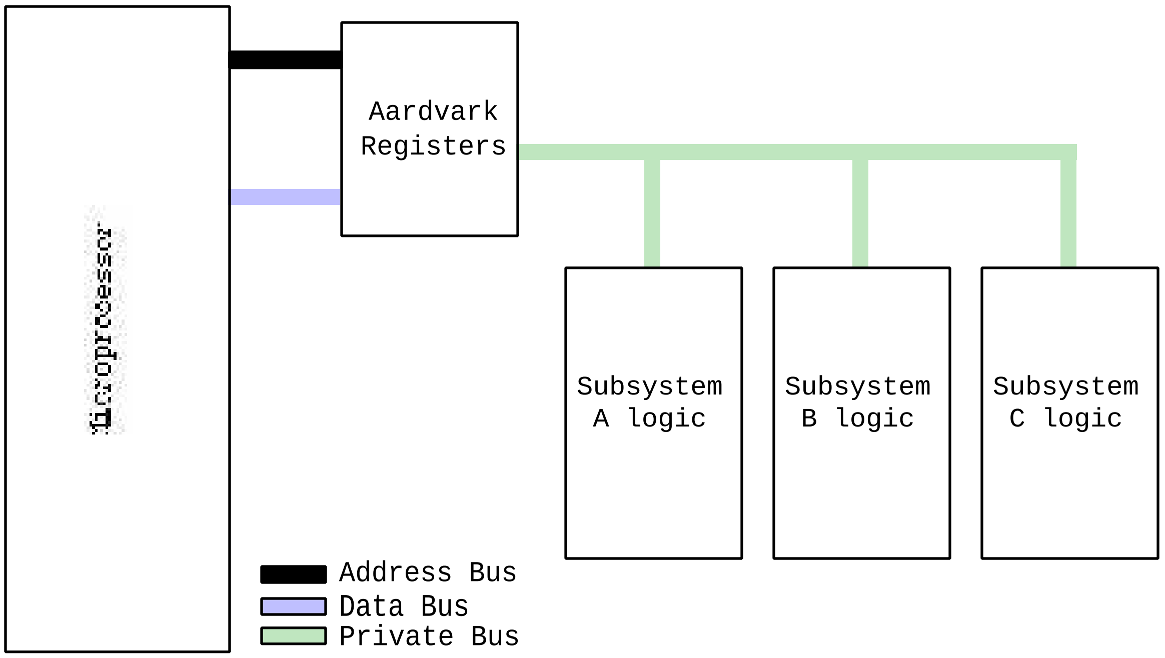


Figure 8 - Main Processor / Subsystem Communication

The main processor communicates via these Aardvark registers using a memory–mapped scheme. Each subsystem has a control and data register assigned to it and monitors its control register waiting for appropriate control information to be presented. Upon being strobed, the subsystem then executes its designated task. During execution, the subsystem may pass control and/or data back to the main processor via the same control and data registers. This technique is the same used with most conventional memory mapped computer subsystems.

4.3. Non–Volatile RAM

4.3.1. Purpose

Aardvark uses non–volatile memory (NVRAM) to store the trusted application baseline data used to verify the integrity of the application executable. As mentioned in the section on subsystem communication, the NVRAM is protected from direct access by the main processor through buffering logic (see Main Processor/Subsystem Communication page 27).

4.3.2. NVRAM Layout

The NVRAM (see Figure 9) is an array of entries comprised of three sections: file signature, file checksum and encryption key. The file signature is a 32 bit value unique to each particular application executable. The file checksum is a 32 bit value which represents a checksum of the entire application executable. The encryption key is a 56 bit value which is used to decrypt each of the segments of the application executable.

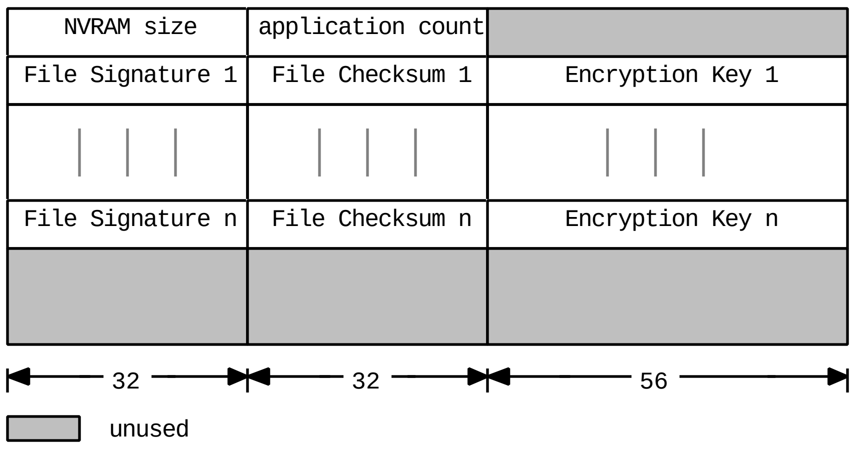


Figure 9 - Non-volatile RAM layout

4.3.3. NVRAM Hardware

The NVRAM could be implemented using the Dallas Semiconductors DE1645EE NVSRAMs (Non–Volatile Static Random Access Memory). These devices provide eight megabits (8Mbx1) of non–volatile storage. Their access time of 70nS makes them ideally suited for this application.

4.3.4. NVRAM Data Elements

4.3.4.1. File Signature

Each Aardvark application would have its own unique set of file signatures. These would provide a mechanism by which document creation could be tracked. As seen above, the size of the application file signature value is 32 bits. This size accommodates the needs of Aardvark to uniquely identify applications and also allows for use outside the scope of the virus–resistant nature of Aardvark.

The application file signature can be seen as a data structure having three main elements (see Figure 10). These elements are a unique application ID, an application type (or class) and a block of application configuration data.

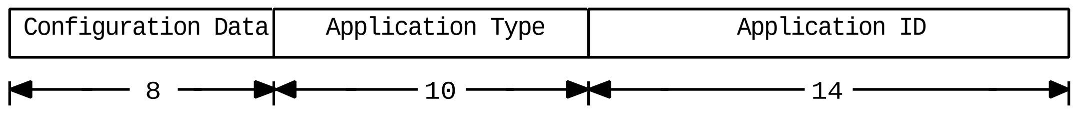


Figure 10 - Application Signature Elements

4.3.4.1.1. Application Type

The application type represents the category in which the application falls. Examples of possible application type include: drawing (object oriented), painting (pixel oriented), word processing, telecommunications, spreadsheet, notebook, game, etc. The field is 10 bits wide allowing for 1024 possible types to be assigned.

4.3.4.1.2. Application ID

Each application which has been released by a commercial manufacturer and is deemed to be ‘trusted’ will have a unique application identification code. This code is 14 bits in length allowing for 16,384 possible applications within a particular application type.

4.3.4.1.3. Configuration Data

The configuration data element within the application signature structure is itself a data structure. This structure is composed of 8 bit flags which provide information as to the structure of the application. The assignment of these flags is shown in Figure 11 on page 32.

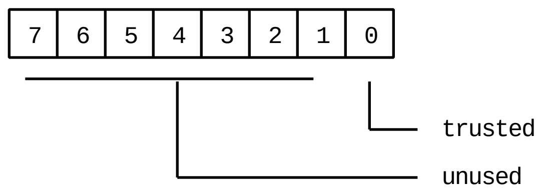


Figure 11 - Configuration
Data Bit fields

The specifics of each bit field is discussed in those sections using them.

4.3.4.1.4. File Signature Administration

The question naturally arises as to how the list of the application types and IDs will be assigned and maintained. This task would be undertaken by the organization for the Aardvark computer architecture itself. This methodology of application signature administration has been demonstrated to work quite efficiently by Apple Computer, Inc. with their Macintosh line of computers. When a new application is created, a file signature is requested and specific file creator and type (the two elements of the file signature) may be requested. Apple Computer, Inc. then responds with file creator and type assignments for the application in question. This system has been in place at Apple since the inception of the Macintosh computer in 1984. In the case of Aardvark–based computer system applications, there is no vendor specific component to the file signature. The vendor would need only specify what type of application they were creating to receive a file signature for the application.

4.3.4.2. Checksum

4.3.4.2.1. Overview

In order to provide a high level application verification, there is a hardware verifiable 32–bit checksum maintained in the NVRAM. It is computed using an algorithm implemented in hardware to reduce the likelihood of reverse engineering and increase speed of checksum computation.

4.3.4.2.2. Operation

When a request is made to the computer system to load an application, a checksum of the application executable is computed and compared with the checksum loaded into the NVRAM. If the two values do not match, an error condition is generated. The user is then informed that the application load request could not be completed since there was apparent corruption of the application executable image (see Application Executable Checksumming page 47).

4.3.4.3. Encryption Key

4.3.4.3.1. Background

Since the trusted application executable is stored in encrypted form, it is necessary to decrypt the individual objects upon loading into memory during execution. The key used for decryption of the application executable as well as the file signature and checksum must be loaded into the NVRAM of the computer during application installation. Two possible methods exist for loading this data. The first is a fully automatic data load and is intended for use with trusted applications. The second is a semi–automatic data load. This section will describe the first method. For information on semi–automatic loading see Semi–automatic Key Loading on page 64.

4.4. Application Executable Decryption Logic

4.4.1. Introduction

As mentioned in earlier sections, Aardvark encrypts the segments of the application executable in order to provide protection from viral incursion. The method of encryption used by Aardvark is the Data Encryption Standard (DES).

4.4.2. Data Encryption Standard

The Data Encryption Standard (DES) is an encryption method developed by IBM and the NSA. It was adopted in 1977 by the National Institute of Standards and Technology as the standard for unclassified U.S. Government applications.

This method of encryption is known as a product cipher. A 56–bit key is used to encipher 64–bit blocks of data through a series of permutation and reblocking operations.

Questions arise as to the security of the encryption [18]. Nevertheless DES is considered adequate for medium security applications. The only known means to decipher enciphered data without the key is via exhaustive search of keys. Since the encryption key used for each trusted application is unique, it is unlikely that an exhaustive search of keys could be performed without significant computer system performance degradation. The encryption key is ensured to be unique through the use of Touch-Memory™ modules (see Touch Memory™ page 35).

4.4.3. DES Hardware

The DES encryption can implemented in hardware with the VLSI Technologies VM007 DES Encryption chip. The speed of enciphering in hardware is 192 megabits per second. The VM007's processing speed should ensure that there will be little to no noticeable system degradation as the application executable segments pass through the chip and into instruction memory. Another reason to implement the encryption mechanism in hardware is to reduce the likelihood of cracking the encryption key via a software routine. The speed of software implementations of DES are 20 kilobits per second on a personal computer and 160 kilobits per second on a VAX [12,13].

4.5. Touch Memory™

4.5.1. Overview

Aardvark uses Dallas Semiconductor's DS1192 Touch Memory™ to ensure the integrity of the encryption key. This device is a self-contained module which when connected to a receiver transmits data in a serial stream. It contains a 48–bit laser etched serial number and 1024 bits of non–volatile memory.

Each Aardvark–based computer system as well as every trusted application will come with its own Touch Memory™ key. This ensures that every application will have a unique encryption key. For details as to how the application executable is paired with a Touch Memory™ key see Application Encryption on page 73.

4.5.2. Touch Memory™ Interface

The Touch Memory™ is interfaced to the NVRAM subsystem through the Touch Memory™ control logic (Kinkajou) (see Figure 12 on page 37).

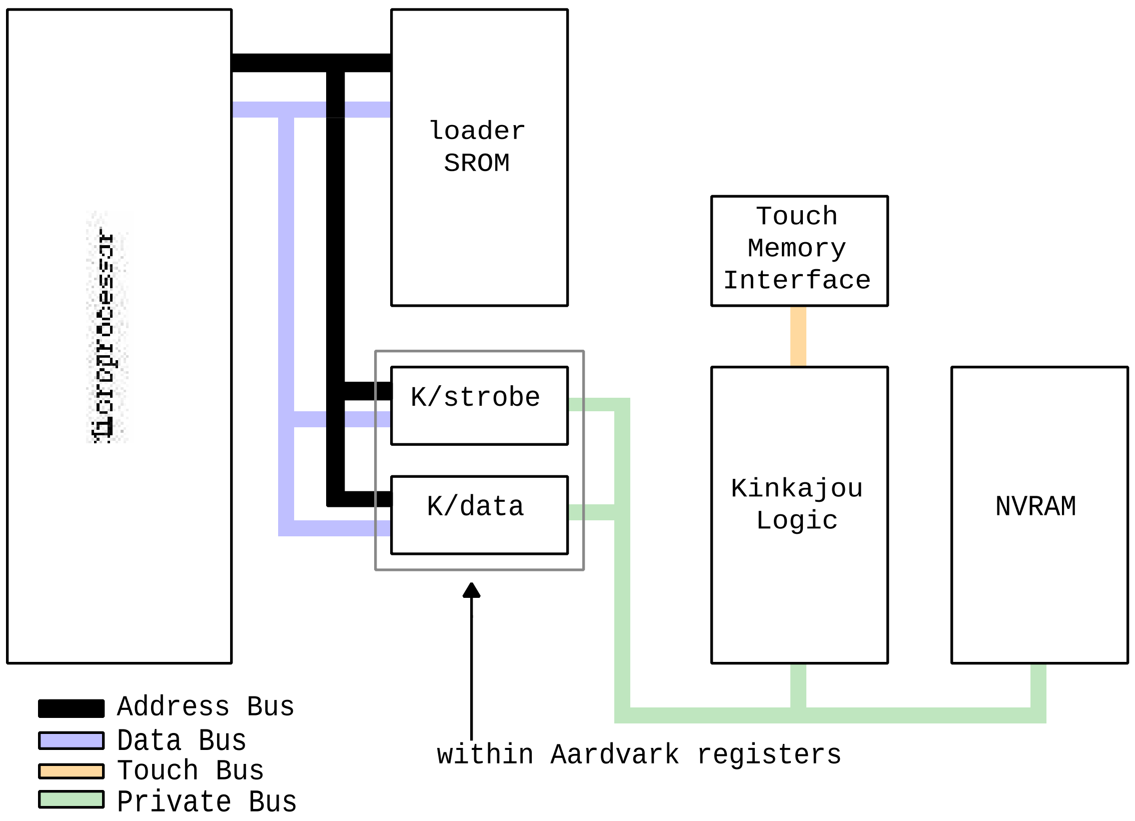


Figure 12 - Kinkajou Interface

When invoked by the installation routines, Kinkajou reads the data from the Touch Memory™ via the Touch Memory™ socket on the front panel of the computer (see Figure 14 on page 41) and transfers it into the NVRAM updating the application count if necessary.

4.5.3. Encryption Key Expansion

The unique ID in the Touch Memory™ module is 48 bits long. DES requires a key 56 bits in length. In order to accommodate the DES key length requirement, the Touch Memory™ key must be extended from 48 to 56 bits by extracting bits from the key and replicating them on either end of the key (see Figure 13). This expansion is implemented in the interface between the Touch Memory™ and NVSRAM. The negative aspect of this method is that it may compromise the strength of the key since the expansion method may be reverse engineered.

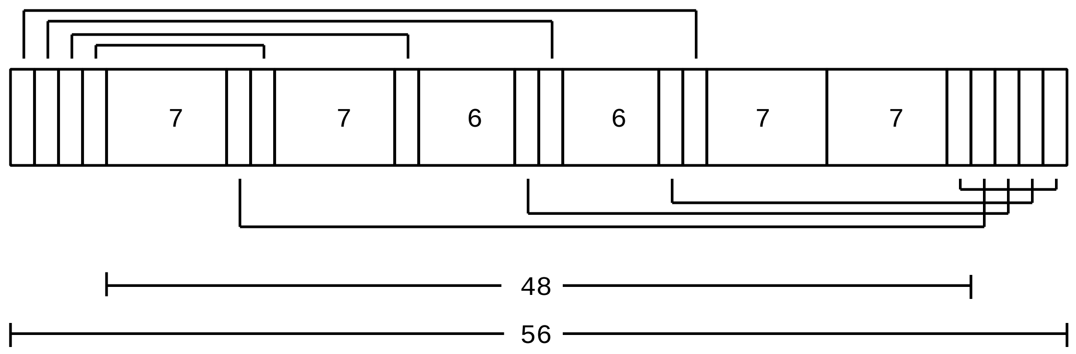


Figure 13 - Encryption Key Expansion

5. APPLICATION LIFE CYCLE

5.1. Introduction

The most important feature of the Aardvark computer architecture is to provide a secure environment for the execution of applications. This section will explore the life cycle of applications as they interact with Aardvark.

5.2. Application Installation

5.2.1. Overview

In order to use an application it must first be installed on the computer system. In the case of Aardvark, this process of installation has two separate paths: one for ‘trusted’ applications and one for non–trusted applications.

5.2.2. Non–trusted Application Installation

In the case of non–trusted applications, the installation process is exactly as it would be on any conventional computer system. The application is transferred from its distribution media into the memory storage of the computer system.

To install a non–trusted application as if it were trusted, the application must be encrypted (see Promoting Applications page 63) and a semi–automatic key load performed (see Semi–automatic Key Loading page 64). These extensions to the Aardvark computer architecture allows for a greater degree of assurance for non-trusted applications.

5.2.3. Trusted Application Installation

5.2.3.1. Introduction

The installation of trusted applications requires additional steps. The signature, checksum and encryption key data must be loaded into the NVRAM (see NVRAM Data Elements page 30). This data is loaded from a Touch Memory™ key (see Touch-Memory™ page 35).

5.2.3.2. Installation Hardware Interface

5.2.3.2.1. Front Panel Interface

An Aardvark-based computer system has some additional hardware mounted on the front panel of the computer for interaction with the user (see Figure 14 on page 41). In addition to the standard reset button and keyboard switch are a mode selector switch and a Touch-Memory™ socket. The functions of these two items will be explained as they are introduced.

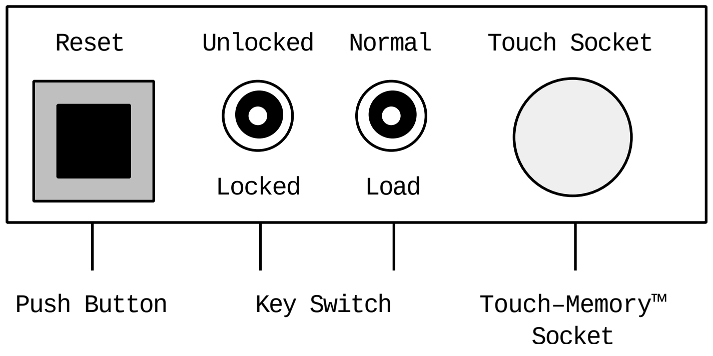


Figure 14 - Aardvark-based Computer Front Panel

5.2.3.2.2. Normal/Load Mode Switch

The installation of trusted applications requires that the user first reboot the computer with the ‘Normal/Load’ key switch in the ‘Load’ position. This switch controls the SROM used by the computer and write access to the NVRAM (see Figure 15 on page 42).

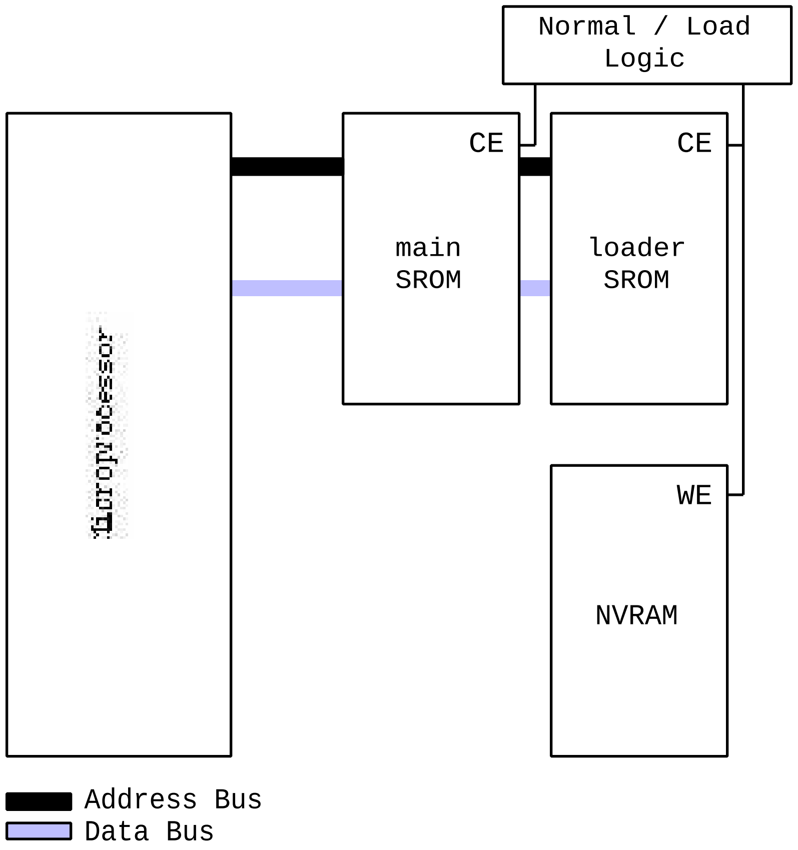


Figure 15 - Normal/Load Switch Interface

It should be noted that the loader SROM is not subject to the restrictions imposed by Packrat (see <u>ROM Code</u> page 59).

The Normal/Load switch provides an additional layer of control over the hardware. It prevents any write access to the NVRAM. The switch is connected to selector logic which when strobed by the processor’s Reset line selects between the normal and load SROMs, and disables or enables the NVRAM for writing (see Figure 16 on page 43). This provides a means by which the machine can switch modes without powering down during mode switches.

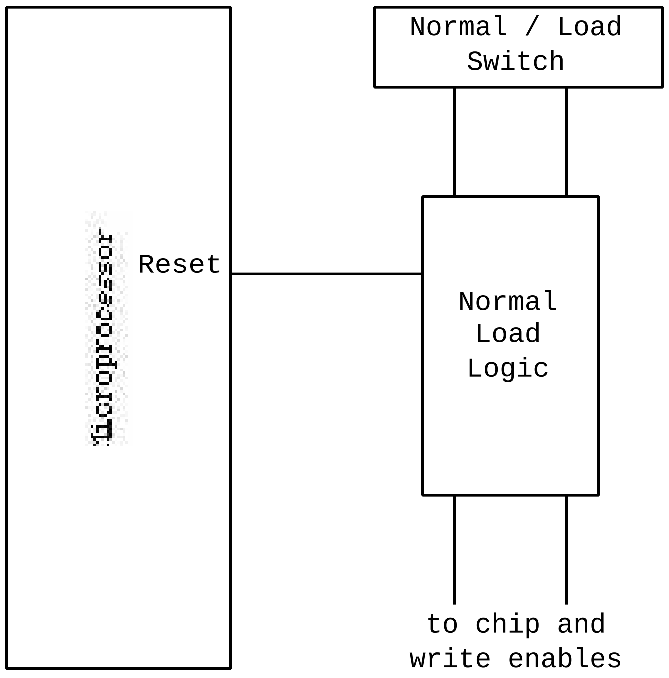


Figure 16 - Normal / Load Logic

5.2.3.3. Installation User Interface

When an Aardvark–based computer system is booted in load mode, the user is presented a simple menu–driven user interface (see Figure 17 on page 44).

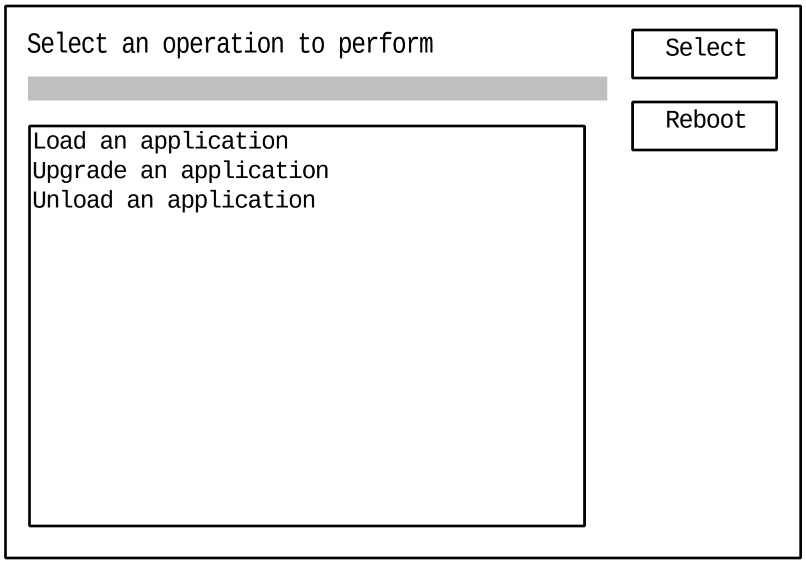


Figure 17 - Top Level User Interface

The user then selects the ‘load an application’ option and is presented with the application installation dialogue (see Figure 18).

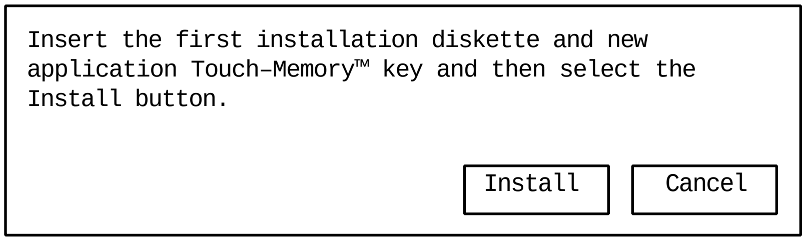

Insert the first installation diskette and new application Touch-Memory™ key and then select the Install button.

Figure 18 - Installation User Interface

If the user confirms the request for installation, the computer system installs the application, its associated configuration files and NVRAM entry. The user may then exit from installation mode and reboot the computer system in normal mode. If the user cancels the request for installation, the top-level interface is again presented (see

Figure 17 on page 44). At this point the user may either select a different application or exit the installation process entirely, as mentioned above.

5.3. Application Execution

5.3.1. Overview

When the user requests that the system load an application for execution, a series of checks are made to determine whether the application is trusted. If these checks are passed without incident, application execution is allowed to proceed. This section will detail the verification process that occurs when an application executable is invoked.

5.3.2. Application Type Checking

Upon a request for application execution, the operating system passes the file signature associated with the application executable and stored in the computer system’s directory structure to the off-processor register used by the application type checking logic (Gopher). Gopher is then invoked by strobing its control address (G/strobe). The operating system then waits for Gopher to respond. The result is loaded into the bidirectional Gopher data register (G/data) and the strobe is tickled (see Figure 19 on page 46).

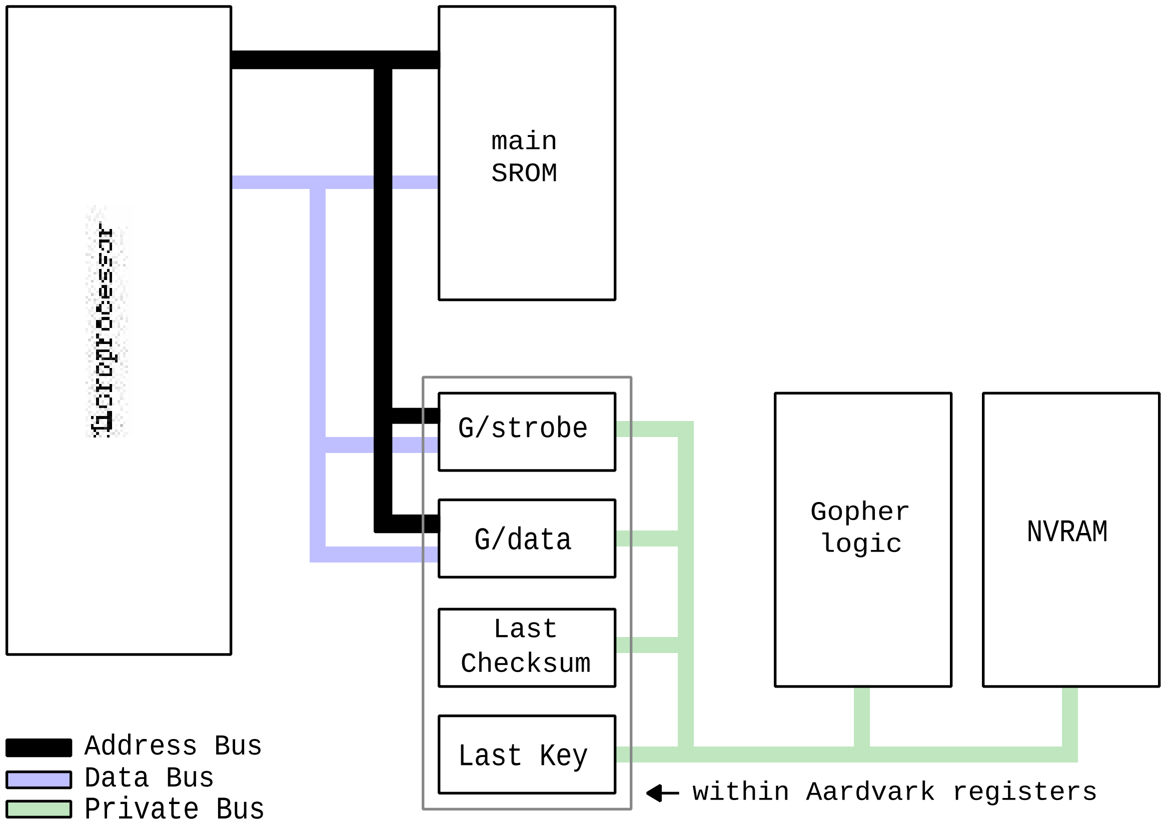


Figure 19 - Gopher Interface

Gopher searches the NVRAM for an entry that matches the one passed to it. If one is found, the NVRAM entries for the decryption key and checksum are transferred to the off-processor registers ‘Last Key’ and ‘Last Checksum’ respectively. A positive return code is loaded into the bidirectional Gopher data register, and the strobe is tickled to let the operating system know that the search has been completed. If the application file signature does not match any in the NVRAM, the ‘Last Key’ and ‘Last Checksum’ registers are cleared, a negative return code is loaded into the bidirectional Gopher data register and the strobe is tickled.

If a negative response is returned from Gopher, the application loader knows that the application is not trusted. If this is the case, the application will be loaded without activating the decryption logic. If a positive response is returned, the application is assumed to be trusted and decryption will take place during segment loading.

5.3.3. Application Checksumming

Once an application has been determined to be trusted, it must be checksummed. As mentioned in the earlier section on NVRAM data elements (see Checksum page 33), an application checksum is stored in the NVRAM. This gross level checksum provides a rapid, if not totally secure, method of assuring, on a high level, that the application has not been tampered with in any blatant manner.

The checksum is performed by hardware logic interfaced via a strobe/bidirectional interface (Cricket). Cricket is invoked by loading the bidirectional Cricket data register (C/data) with the first word of the application executable and then strobing its initiate address (C/strobe) . Cricket initializes the checksum to a predetermined initialization values and reads the data in C/data (see Figure 20 on page 48). Cricket then acknowledges the data and waits for additional data to be sent.

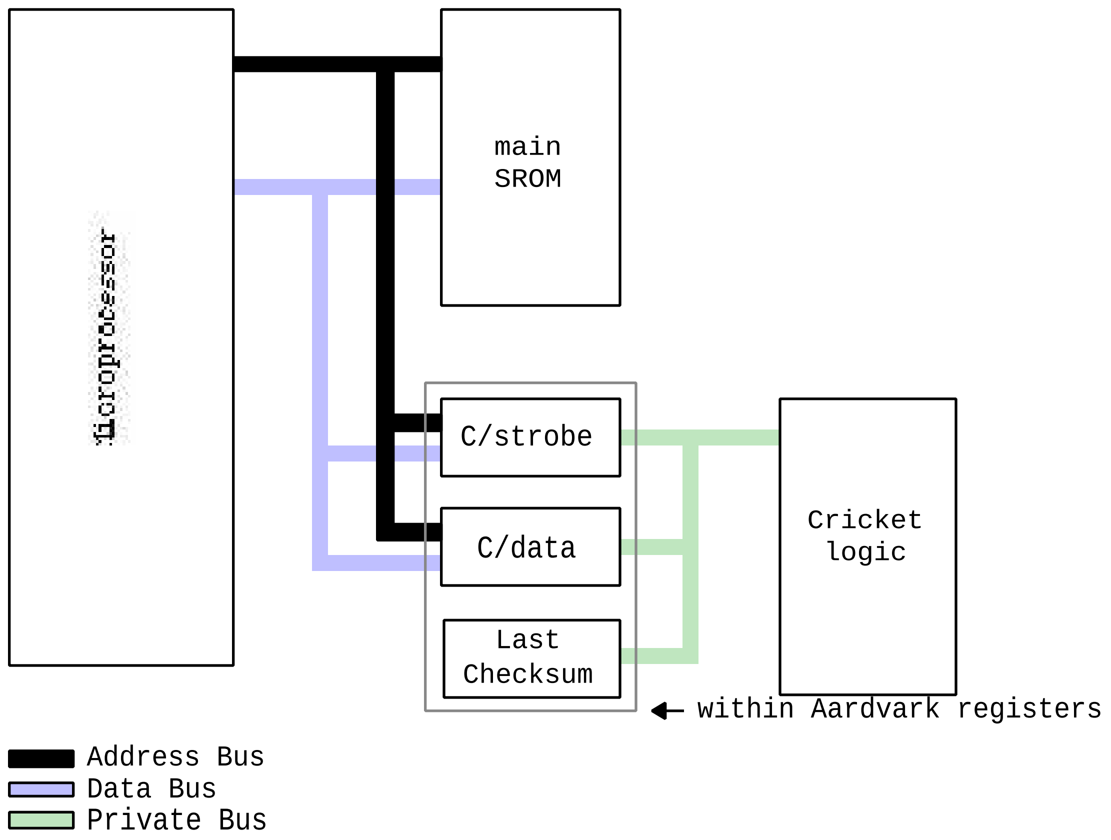


Figure 20 - Cricket Interface

When the entire application has been passed to Cricket in this manner, the operating system signals that it has completed the loading to Cricket via C/strobe. Cricket then compares the checksum which it has computed against the value in the last checksum data register. If the two match, a positive status code is sent back to the operating system via C/data. Otherwise, a negative status code is generated and returned.

If a negative status code is returned from Cricket, the user is informed that the application executable has been corrupted and should be reloaded. If a positive status code is returned from Cricket, the actual loading of the application into memory is allowed to begin.

5.3.4. Application Executable Loading

5.3.4.1. Overview

Once the application executable has been verified as being trusted and having a valid NVRAM checksum, the operating system will initiate an executable load. As mentioned earlier, Aardvark’s chief method of preventing application executable viral infection is application executable segment encryption (see Aardvark’s Computer Architecture page 24). If a non–trusted application has been requested to be loaded, it may do so without having to pass through the decryption logic (Dolphin).

5.3.4.2. Non–trusted Segment Loading

As mentioned above, if a non–trusted application is being loaded, the segments do not pass through Dolphin. They instead are loaded directly into instruction memory by the segment loader.

5.3.4.3. Trusted Segment Loading

Before any segment of a trusted application executable is loaded into instruction memory is passes through Dolphin for decryption (see Figure 7 on page 25). Once decrypted the segment is loaded into instruction memory. After all the initial load segments of the application executable have been loaded, the application will then be available for execution.

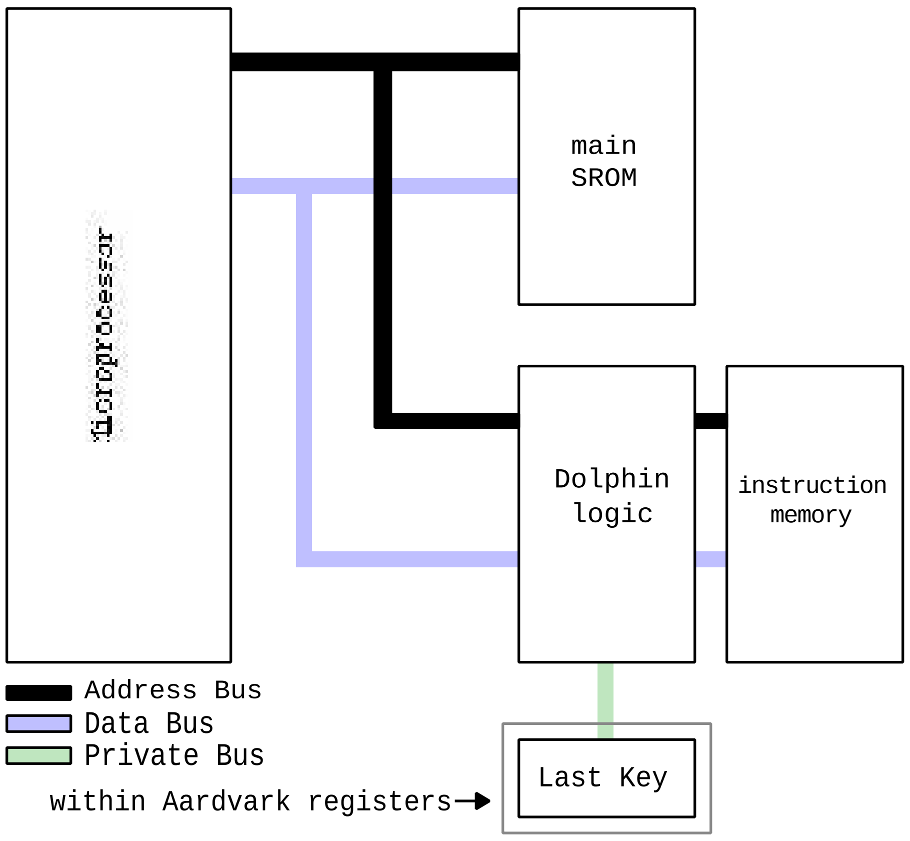


Figure 21 - Dolphin Interface

5.3.5. Segment Swapping

If the need arises for a segment of an application to be swapped, the displaced segment is not swapped out to a high speed swapping memory unit but rather overwritten. Any time a new segment is required, it is loaded from disk. This prevents any access to segments that are in memory in a decrypted form and strengthens the protection against modification of in memory code. The handling of in memory segments is discussed further in the section on separate instruction and data memory handling (see <u>Physically Separate I and D</u> page 56).

5.4. Application Deinstallation

5.4.1. Overview

When the user determines that an application has reached the end of its usefulness, the user may decide to deinstall the old application and replace it with a new one.

5.4.2. Non–trusted Application Deinstallation

The deinstallation of non–trusted applications is accomplished in the same way as on non–Aardvark computer systems. The application is simply deleted from the computer system’s storage along with any configuration files which may accompany it.

5.4.3. Trusted Application Deinstallation

5.4.3.1. Introduction

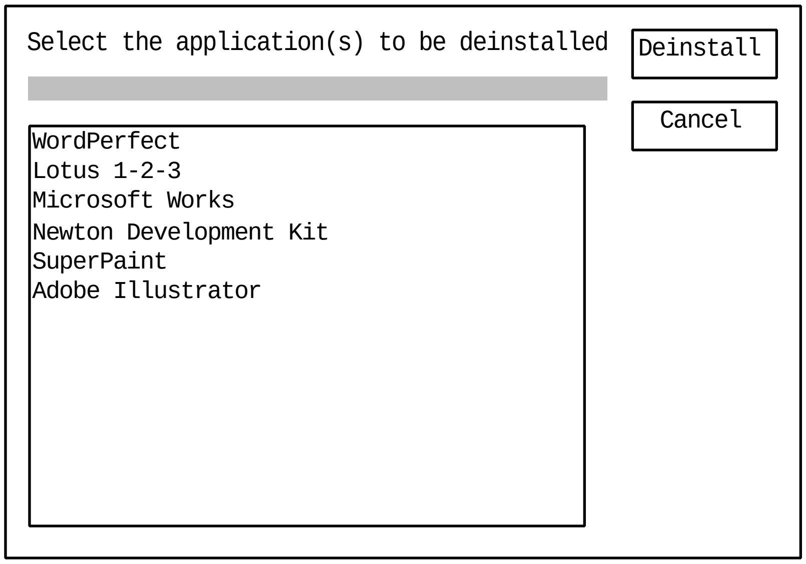
The deinstallation of a trusted application requires not only that the application executable and associated configuration files be removed from the storage of the computer systems, but also that the NVRAM entry relating to the application be removed.

5.4.3.2. Deinstallation Hardware Interface

To deinstall an application and remove the NVRAM entry, the computer system must be in load mode. This is accomplished by selecting the ‘load’ position of the ‘Normal/Load’ switch and rebooting the computer system (see Installation Hardware Interface page 40). The user then instructs that the trusted application, its associated configuration files and NVRAM entry be removed from the computer system (see Deinstallation User Interface page 52). The computer system may then be rebooted in normal mode.

5.4.3.3. Deinstallation User Interface

Once the computer system is booted into load mode, the user is presented with the top level menu–based interface (see Figure 17 on page 44). The user then selects the ‘deinstall application’ option and is presented with a list of trusted applications from which to choose (see Figure 22 on page 53).

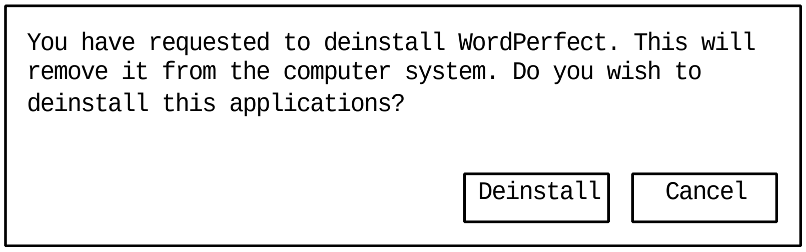


Select the application(s) to be deinstalled

WordPerfect
Lotus 1-2-3
Microsoft Works
Newton Development Kit
SuperPaint
Adobe Illustrator

Figure 22 - Deinstallation User Interface

Upon choosing one or more applications to deinstall, a dialogue is presented requesting that each request to deinstall an application be confirmed (see Figure 23).



You have requested to deinstall WordPerfect. This will remove it from the computer system. Do you wish to deinstall this applications?

Figure 23 - Deinstallation Confirmation Dialogue

If the user confirms the request for deinstallation, the computer system removes the NVRAM entry, its associated configuration files and finally the application itself. The user may then exit the deinstallation mode and reboot the computer system in normal mode. If the user cancels the request for deinstallation, the main deinstallation dialog is again presented (see Figure 22). At this point the user may either select a different application or exit the deinstallation process entirely, as mentioned above.

5.5. Application Upgrading

5.5.1. Overview

It is the rare exception in the software industry when an application never requires upgrading with either bug fixes or new features. The Aardvark computer architecture allows for this contingency in a straightforward manner.

5.5.2. Non–trusted Application Upgrading

A non–trusted application is upgraded on an Aardvark–based computer system in the same manner that it would be if it were on a non–Aardvark–based one. The application executable and any pertinent support are removed and replaced with the upgraded versions of the same (see also Non–trusted Application Installation page 39).

5.5.3. Trusted Application Upgrading

Trusted applications must be upgraded through the same mechanism used for installation and deinstallation of trusted applications. The computer system is first booted in load mode (see Installation Hardware Interface page 40). Next the user selects the upgrade trusted application option from the top–level installation dialogue (see Figure 17 on page 44). The application upgrade dialogue then appears (see Figure 24 on page 55).

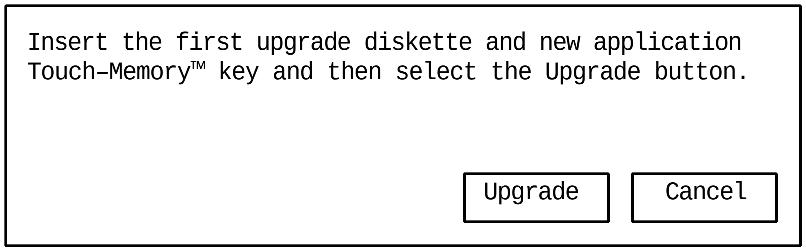

Insert the first upgrade diskette and new application Touch–Memory™ key and then select the Upgrade button.

Figure 24 - Upgrade Confirmation Dialogue

The user is instructed to insert the first upgrade diskette and new application Touch Memory™ key. The user may then select the ‘Upgrade’ button to proceed with the upgrade process or the ‘Cancel’ button to abort the process. If the user chooses to proceed, they will be prompted if additional diskettes are required during the upgrade process. Upon completion of the upgrade process, the user is returned to the top–level installation dialogue. At this point the user can reboot the computer system in normal mode. If the user selects to cancel, they are returned to the top–level installation dialogue.

6. SYSTEM INTEGRITY

6.1. Introduction

Since a computer system is not merely a collection of hardware, steps must be taken to insure the security of the code during its execution. As seen earlier, the modification of operating system code and the corruption of file structures by accessing of low–level SROM calls represent serious threats to applications and the operating system. This section of this document addresses these areas.

6.2. Code Integrity

It is one thing to protect the distribution of an application executable from viral infection and quite another to ensure that there is no contamination while it is in execution. The former is dealt with through the encryption of the application executable’s segments. The latter is handled through the use of several layers of protection.

6.2.1. Physically Separate I and D

A lack of memory has traditionally been a problem within the community of computer users. In the early years of computing this was mainly a hardware issue. At first it was simply not possible and later not cost effective to provide the physical memory required for large programs and large in–memory data structures. Additionally, as technology progressed, the definition of what constituted a large program became ever bigger.

In order to cope with the unending requirements for space, several techniques were developed. One of the most significant of these was that of executable segmentation and paging. This provided a mechanism which allowed execution of applications with executables larger than the physical address space would provide. On the hardware side came physically separate address spaces for the application executable’s code (instructions) and the variables used by the application executable (data). This effectively doubled the address space of the computer with minimal additional control logic. This method of separating the address space into two physical sections for instructions and data was called the ‘[physically] separate I and D’ model.

As computer technology matured, techniques were developed which allowed efficient handling of application executables that required greater address space than the computer could provide. The most well known of these techniques is virtual memory management. Simply put “virtual memory is memory that you think you have, but don’t.” This allowed for applications which did not have to use segmented data spaces, greatly simplifying the handling of large data structures especially dynamically created ones.

These advances in address space management made physically separate instruction and data space unnecessary.

The combination of hardware and software memory management made it possible to handle most situations quite gracefully.

Currently most computer memory systems are designed with a single physical address, referred to as a ‘[physically] combined I and D’ model.

6.2.2. Application Executable Protection

Aardvark’s Computer Architecture (see Figure 7 on page 25) uses the separate I and D model. This memory implementation model was chosen to prevent modification of application executables while they were in an unencrypted state in the active memory of the microcomputer. Unlike the traditional implementation of the memory separate I and D model, Aardvark does not allow non–segment loader writing to the Instruction address space. Additionally, the Instruction address space cannot be read from (this includes the ROM address space [see <u>ROM Code</u> page 59]) except by the instruction loader. This greatly inhibits any attempt to determine the method of encryption by comparing encrypted and unencrypted versions of the same segment. This inhibition factor of Aardvark may be nullified by the optional encryption extension to the architecture (see <u>Deterrent</u> page 68 and <u>Encrypting User Applications</u> page 62).

6.3. ROM Code

The ROM code, that code which is resident in the SROM, is called by various applications during the normal operation of a computer. In conventional computer architectures, this code may be accessed by any application, both in the sense of subroutine calling and also of reading the data stored in the ROM itself. The open nature of this type of design allows any application to access such functions as directory structure manipulation routines, low level disk routines, etc. Viruses exploit these operations heavily in order to invade computer systems.

Aardvark deals with this issue with logic which acts as a checkpoint between the caller of SROM routines and the SROM routines (Packrat). The SROM is partitioned into two addressable areas (see Figure 25).

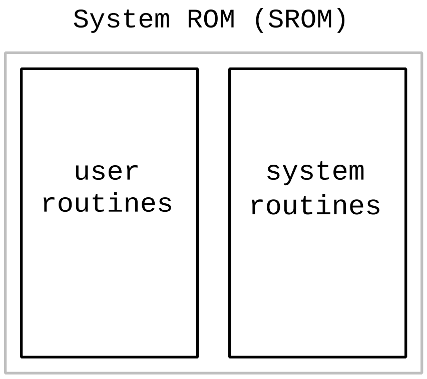


Figure 25 - SROM partitioning

When a call is made to a routine resident in the SROM ROM, that address is hardware verified to determine its origin. If the routine has been designated as a user routine for general use, the routine is allowed to execute without question. If however, the routine has been designated as a system routine, then a check is made to verify that the caller is a user level or another system level ROM routine (see Figure 26). If so, the routine is allowed to execute. If not, an error condition is generated. This error condition takes the form of an interrupt generated by Packrat which is addressed by the SROM and reported to the user via an error dialogue.

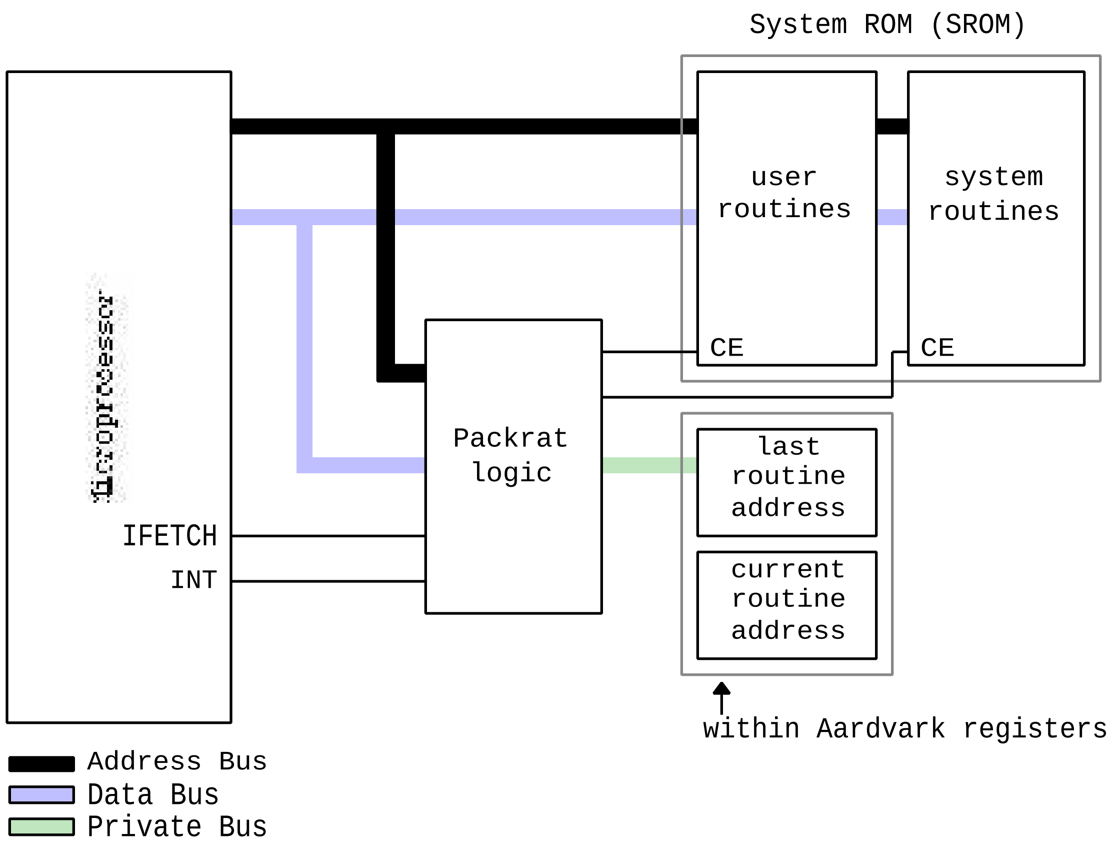


Figure 26 - Packrat Interface

The actual implementation of Packrat is simple. The ‘last routine address’ is a transparent latch holding the address of the last program counter value. It is loaded from another transparent latch (current routine address) containing the current value of the program counter. When an instruction is accessed, the instruction’s address is put onto the address bus and an instruction fetch sequence is indicated on the processor’s control lines. The last routine address latch loads the value of the current routine address latch, and the current routine address latch loads the value on the address bus. The current routine address is then compared with a the system ROM routine address range and if it falls outside of that range, the chip enable logic is asserted. If the current routine address falls within the range of the system ROM routines, the last routine address is checked to verify that it falls within the range of the SROM proper. If it does, the chip enable logic is asserted. If not, an error is generated in the form of an interrupt.

The Packrat does not need to be separate from the address decoder. It is illustrated in this manner for clarity. In actual implementation, it would be best to have the verification logic integrated with the address decoder logic for speed considerations.

7. ARCHITECTURAL EXTENSIONS

7.1. General Comments

Aardvark is designed to be as open as possible while providing the maximum protection to applications. This section discusses architectural extensions which may be made to the system to provide additional features which some users may consider useful. It should be noted that any modifications to the architecture may create situations which compromise the systems ability to protect the encryption mechanism.

7.2. Encrypting User Applications

7.2.1. Overview

Aardvark may be extended to allow for the encrypting of user applications allowing the user to take existing applications which are not inherently ‘trusted’ and promote them to such a status.

Since this extension to the architecture would make it possible for the direct comparison of encrypted and non–encrypted executable code segments, a situation is created in which the encrypting mechanism of the architecture may be compromised. This is, of course, dependent upon the encrypting mechanism used. There should be no problem with systems such as DES, but other mechanisms for encryption may be more susceptible to this type of reverse engineering.

7.2.2. Promoting Applications

In order to promote an application to trusted status, additional hardware and software need to be introduced to Aardvark’s architecture. Support logic (Ferret) which bridge into Cricket, Dolphin and Kinkajou must be added (see Figure 27).

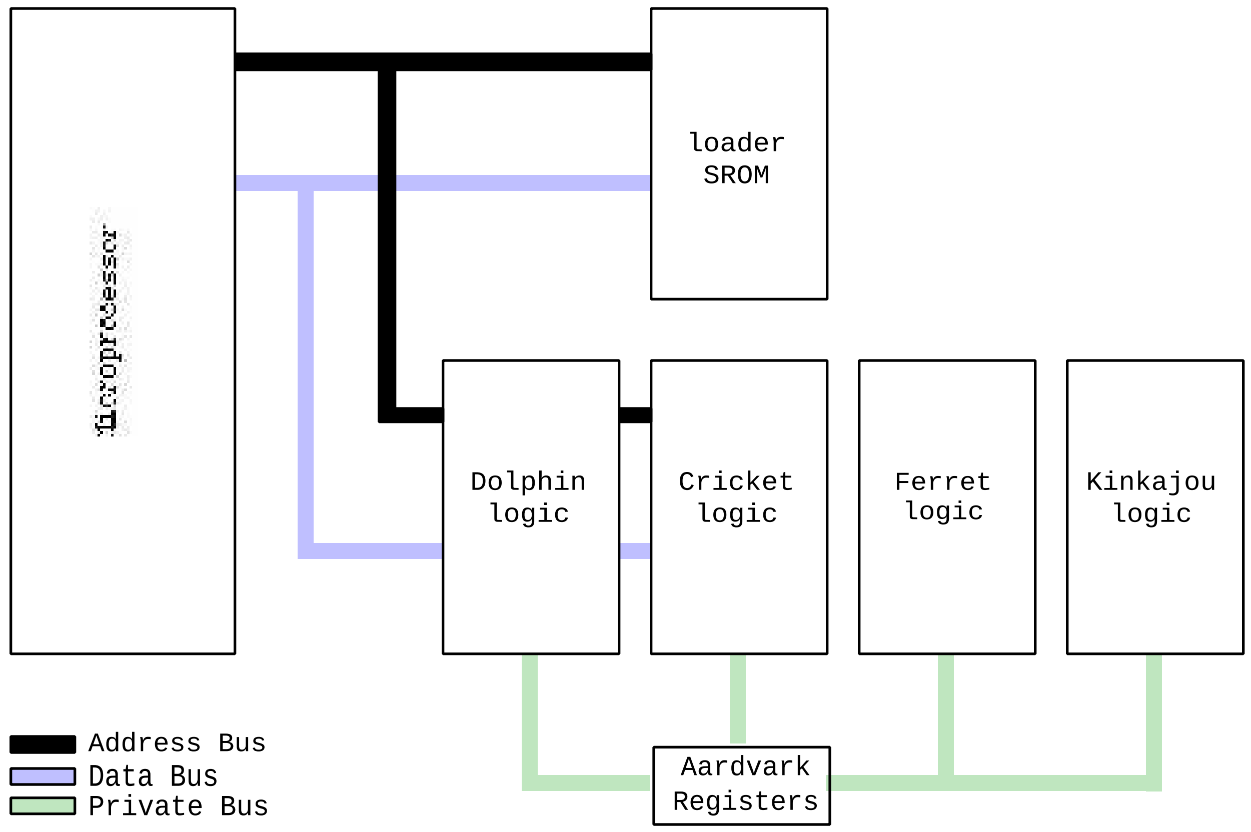


Figure 27 - Executable Encryption Extension

The loader SROM must also be modified to take an existing application executable and transfer it to diskette after processing it through Cricket. Additionally, code must be added to the loader SROM to update the user interface to include this feature.

When an application is promoted, the user is informed of the application signature and checksum. This information is used when the application is installed. As mentioned earlier the file signature is 32 bits wide. The first bit (bit 0 [see Figure 11 on page 32]) indicates the type of file signature (see Table 1).

0	trusted application
1	promoted application

Table 1 - File Signature Configuration Bit 0

Since the operation of application promotion is handled using the loader SROM, it is possible that the user would like the actual installation to be handled at the time of promotion. This would require only code bridges between the promotion and semi–automatic key loading.

7.2.3. Semi–automatic Key Loading

The decryption information required by the system cannot be fully provided by the Touch Memory™ key when installing an application into the system which has been promoted to the status of trusted. Whereas in the case of the fully automatic load, the Touch Memory™ key transfers the decryption key, file signature and file checksum, the semi–automatic load requires that the file signature and checksum be entered by hand during the installation process.

This option would require additional code in the loader SROM. The additional code would be minimal as the only change would be the passing of the signature and checksum to the

NVRAM parallel loader. The hardware modification required would be simply the addition of two memory–mapped registers which would be gated into the NVRAM parallel loader after the Touch Memory™ data had been loaded, thereby overriding the default data.

7.3. Secure Distribution Media

7.3.1. Overview

The integrity of an application executable is dependent upon all the intermediate forms it exists in before it is installed on an Aardvark–based machine. The weakest link in the distribution chain is that of the distribution media. Software may be distributed in two ways so as to provide the highest level of confidence in the distribution.

7.3.2. CDROM

The CDROM is very quickly becoming the most common and popular form of distribution for large software packages. Capable of holding 600MB on uncompressed data, the CDROM is not sensitive to magnetic fields, requires little space and most importantly, is not subject to modification. The main drawback is that the media is not recyclable.

7.3.3. PCMCIA

PCMCIA is a group which has created an international standard for platform independent computer peripherals. These devices include everything from memory expansions to cellular modems to silicon disks. One type of memory expansion is a PC CARD containing only a ROM. As with CDROMs, PC CARD media cannot be modified by normal means (i.e., through software), but can be recycled if the PC CARD’s ROM is actually a EEPROM. In this way, upgrades would not lead to a constant accumulation PC CARDs and their associated Touch Memory™ keys. The old key and CARD can be returned for reuse.

7.4. Data File Security

Aardvark may be extended to provide a stronger data file security. The system SROM may have the file handling routine segregated in such a way as to require that file deletion be handled either by the application which created the data file or from the user interface only. Other applications would be free to read information from these data files but not modify or destroy them. This would effectively prevent an application infected with a virus from attacking data files present on the computer system.

When a request is made to open a file for write or delete access, the data file creator ID would be checked against the current application file signature file ID sub-field. If the IDs matched, the operation would be allowed to proceed. If they did not match, the file request would be denied and the user would be informed via an error dialogue. Since non–trusted applications have no associated file signature, they would automatically be denied write and delete access.

This mechanism does not prohibit the free flow of data within the computer system. It does, however, add the possible requirement that the user manually delete data for which he/she does not have the creator application. Since most data desired for import will be in a standard format, it should require little effort to import the data with a resident application and save the data to a file created by the importing application.

8. SOFTWARE PIRACY

8.1. Summary Statement

Several questions arise regarding the issue of software piracy as it relates to my proposed architecture. I will address them in this section.

8.2. Scope

The scope of this proposed architecture does not deal with the possibility that ‘trusted’ distribution media have been tampered with. If a piece of software has been ‘cracked’, modified, re–encoded and redistributed there is little that this architecture can do to help. The architecture will deal with this by preventing the spread of any virus through the same means which it uses to prevent such a spread when a ‘non–trusted’ product containing a virus is introduced to the system.

8.3. Deterrent

Aardvark, by its very design, discourages piracy. There will no doubt be a minority population in the computer community who will see this architecture as yet another technical challenge to surmount. As mentioned in earlier sections dealing with the specific implementation of the segment encoding, it is intended to be very difficult to decrypt the application programs loaded onto a computer using this architecture. I do not consider this to be a serious threat to the security of the machine. On the contrary, I would think that the incidence of trusted application infection and theft would decrease, since there is no way to retrieve a decryption key from a machine once it has been loaded.

If an individual copied a trusted application from a machine based on Aardvark and loaded onto one not based on Aardvark, the application would not execute since there would be no mechanism for on–the–fly decryption. Similarly, if the application were loaded it onto an Aardvark–based machine, it would still not run since the decryption key for that particular copy of the application would not be present. As explained in an earlier section, it is not possible to load the encryption key manually or for that matter retrieve it to a human readable form without highly specialized hardware. Even if the key were cloned, it would be highly prohibitive to create and distribute.

Assuming that both the software and key are stolen, the owner need only report the loss to the manufacturer. If any problems arise, the individual would be identified immediately via the software’s serial number. There is a high degree of disincentive toward mass installation of an application on a network.

It is also possible to protect a non–trusted application from piracy. Any application may be encrypted using the ID key which would come with each Aardvark–based computer (see

<u>Encrypting User Applications</u> page 62). Once this has been accomplished the application may be transferred to diskette and installed as if it were a trusted application.

9. Trusted Application Development

9.1. Introduction

In the creation of any new computer system, let alone a completely new computer architecture, the question is, “Is there software available?” The numerous failed computer systems show that the availability of computer software is a critical issue. The Aardvark computer architecture is designed in such a way that it is very easy to create an application for it or port an existing application to it.

9.2. Trusting Trust

How can you ensure that the trusted applications created for a computer architecture which is highly virus–resistant are free of viruses? Ken Thompson addresses this issue of trust in his 1983 ACM award speech [14].

Basically, the problem is one of trust. It is impossible to be 100 percent certain that an application executable not personally assembled is not suspect. The task is to ensure a high enough level of certainty so that there is confidence that application development tools are free from viruses and that the output of these tools are also.

It is presumed that the development tools used on an Aardvark–based computer system are themselves trusted. If this assumption is true, then no problems should arise with any application produced by these development tools.

9.3. Application Creation

In order to create an application which can be run on an Aardvark–based computer system, the most important rule to follow is: Thou shalt not write self–modifying code. As seen in the section on physically separate I and D (see <u>Physically Separate I and D</u> page 56), it is imperative that application be written is such a way as to not use this technique of coding, since the application executable code is not accessible by the application itself.

A second rule to follow is: Thou shalt not imbed data in code segments. Since the instruction and data memory spaces are physically separate, there is not way to access any data embedded within the application code.

Aardvark requires well disciplined and modular programming, both in terms of the code and data structures. This programming philosophy can already be seen to a large degree in the Apple Macintosh application environment. Although there is not hardware support for it, there is a clearly defined separation of code and data, and also a separation of pure data (akin to flat files) and structured extensible data (i.e., code and data resources).

As mentioned in the section on file signatures (see <u>File Signature Administration</u> page 32), the individual application types and IDs would be assigned by the organization maintaining the Aardvark computer architecture. A company would need only request a new file signature based on the application type.

9.4. Application Encryption

Once an application has been created for or ported to an Aardvark–based computer, it is necessary to encrypt the application executable prior to distribution. This process is similar to that explained in the section on application encryption (see <u>Encrypting User Applications</u> page 62). The only difference is that, in the production environment, the duplication of large quantities of the application would require a mechanical mechanism to feed the Touch Memory™ keys to the system for programming and additional software in the loader SROM to handle such mass duplication.

9.5. Application Distribution

Distribution of Aardvark–based computer application software is accomplished in the same manner as non–Aardvark–based software. As seen from the section on application installation (see <u>Application Installation</u> page 39), the only additional material needed to be packaged and distributed is the Touch Memory™ key. All other materials and channels of distribution remain the same.

10. CONCLUSIONS AND RECOMMENDATIONS

10.1. Conclusions

The Aardvark architecture is one that addresses the issue of viral instantiation into application executables. Its additional protection of low level system routine strengthens the protection of the computer system as a whole. In most cases this is all that is necessary to prevent the spread of viruses.

The problem of data file infection is not dealt with extensively. There is, however, provision within the scope of the architecture for addition of mechanisms which would disallow access to data files by all but trusted applications. This is briefly discussed in the section on extensions.

In the final analysis, the effectiveness of this design proposal cannot be fully evaluated unless a prototype is constructed. When viewed in isolation and together, the elements of this architecture lead to the conclusion that the architecture is sound and will indeed act as an effective deterrent to computer viruses.

10.2. Recommendations

There is sufficient material presented in this document for Aardvark to be implemented by a development team within a period of 18 months. I would recommend that such action be taken. My reasoning for this is two–fold. First, I would like to see the new concepts in virus–resistant computer architecture exploited and put into the public sector. Second, greater exploration of this type of architecture is needed. The implementation of the architecture would lead to additional and possibly more comprehensive solutions to the ongoing problem of computer viruses.

As with the introduction of any new technology, there will be a large initial investment on the part of those who decide to pursue this architecture. Although the hardware needed to modify an existing microcomputer–based architecture are minimal (additional control logic, additional SROMs, NVRAM, etc.), a paradigm shift will be needed on the part of both the computer industry and the computer user. At the present it is considered sufficient to periodically scan for viruses and recover from the attacks which occur. The computer community must come to see that if virus–free environments are to exist, computers must be designed to withstand viral attacks. This paradigm shift is akin to the one which occurred regarding graphical user interfaces (GUIs). At one time GUIs were considered a curiosity. They are now not only a reality, but a standard feature which we have come not only to appreciate but expect.

REFERENCES

R.1. General References

Anderson, Ian. “Viral invader spreads havoc in American computers.” New Scientist 120 (12 November 1988): 24.

“Army to award contract for studying potential of computer viruses as electronic countermeasure.” Aviation Week 132 (14 May 1990): 38.

Barron, Janet. “Two Mac viruses.Byte 14 (June 1989): 278.

“Beware of vandalware.” IEEE Spectrum 28 (February 1991): 66.

“Carleton University Hi–Tech Update ‘88.” IEEE Communications 27 (May 1989): 74.

Chapman, Gary. “CSPR statement on the computer virus.” CACM 32 (June 1989): 699.

Cipra, Barry. “Eternal plague: computer viruses.” Science 249 (21 September 1990): 1381.

“Computer Viruses ‘89.” Datamation 35 (1 April 1989): 60–1.

“Cornell issues report on computer worm.” Computer 22 (June 1989): 99.

Crawford, Diane. “Two bills equal forewarning.” CACM 32 (July 1989): 780–2.

Denning, Peter. “Computer viruses.” American Scientist 76 (May/June 1988): 236–8.

Denning, Peter. “The Internet worm.” American Scientist 77 (March/April 1989): 126–8.

Dewdney, A. “Computer recreations; of worms, viruses and Core War.” Scientific American 260 (March 1989): 110–3.

Diehl, Stanford. “Rx for safer data.” Byte 16 (August 1991): 218–24+.

Dutton, Gail. “At the edge of chaos: artificial life?” IEEE Software 9 (January 1992): 88–9.

Eisenberg, Ted. “The Cornell Commission: on Morris and the worm.” CACM 32 (June 1989): 706–9.

Fainberg, Tony. “The night the network failed.” New Scientist 121 (4 March 1989): 38–42.

Farber, David. “NSF poses code of networking ethics.” CACM 32 (June 1989): 688.

Ferbrache, David. A Pathology of Computer Viruses. London: Springer–Verlag, 1992.

Flynn, Jennifer. 20th Century Computers and How They Worked. Carmel, Indiana: Alpha Books, 1993.

Fox, Barry. “Computers get stoned on patent discs.” New Scientist 131 (10 August 1991): 24.

Greenberg, Ross. “Know thy viral enemy.” Byte 14 (June 1989): 275–80.

Hilton, Phil. “IBM fails to squash ‘virus’ scare.” New Scientist 118 (14 April 1988): 34.

Hirst, Joe. “Rotten to the core: bombs, Trojans, worms and viruses.” New Scientist 121 (4 March 1989): 40–1.

Hodges, Parker. “The viral age.” Datamation 34 (1 December 1988): 96.

Holden, Constance, ed. “Rogue AIDS disk alarms researchers.” Science 247 (5 January 1990): 24.

“How deadly is the computer virus?” Electrical World 202 (July 1988): 35–6.

Joyce, Edward. “Software viruses: PC–health enemy number one.” Datamation 34 (15 October 1988): 27–8+.

Kocher, Bryan. “A hygiene lesson.” CACM 32 (January 1989): 3+.

Lefohn, Allen. “The computer virus.” JAPCA 38 (September 1988): 1102.

Lerner, Eric. “Computer virus threatens to become epidemic.” Aerospace America 27 (February 1989): 14–6+.

Marshall, Eliot. “The scourge of computer viruses.” Science 240 (8 April 1987): 133–4.

Marshall, Eliot. “Worm invades computer networks.” Science 242 (11 November 1988): 855–6.

Marshall, Eliot. “The worm’s aftermath.” Science 242 (25 November 1988): 1121–2.

McAfee, John. “The virus cure.” Datamation 35 (15 February 1989): 29–40.

McLean, John. “The specification and modeling of computer security.” Computer 23 (January 1990): 9–16.

Mickle, Marlin. “A holistic response to viruses.” IEEE Micro 9 (June 1989): 89.

Nordwall, Bruce. “Rapid spread of virus confirms fears about danger to computers.” Aviation Week 129 (14 November 1988): 44.

Preiss, Ralph. “Position paper on computer viruses planned.” Computer 22 (February 1989): 82.

Rochlis, Jon. “With microscope and tweezers: the worm from MIT’s perspective.” CACM 32 (June 1989): 689–98.

Saeed, Faisel. “International Microcomputer Software Inc.” Computer 24 (October 1991): 86–7.

Saffo, Paul. “Consensual realities in cyberspace.” CACM 32 (June 1989): 664–5.

Schlack, Mark. “How to keep viruses off your LAN.” Datamation 37 (15 October 1991): 87–8+.

Seeley, Donn. “Password cracking: a game of wits.” CACM 32 (June 1989): 700–3.

Shulman, Seth. “(Artificial) germ warfare.” Technology Review 94 (October 1991): 18–9.

Spafford, Eugene. “Crisis and aftermath (Internet worm).” CACM 32 (June 1989): 678–87.

“Time bomb ticks in computer networks.” New Scientist 124 (21 October 1989): 26.

Waldrop, M. “PARC brings Adam Smith to computing.” Science 244 (14 April 1989): 145–6.

Wallich, Paul. “Hostile takeovers: how can a computer welcome only friendly users?.” Scientific American 260 (January 1989): 22+.

“Wanted: computer virus antidote.” Design News 44 (19 December 1988): 36.

Watts, Susan. “‘Health campaign’ needed to beat computer virus.” New Scientist 121 (21 January 1989): 26.

Watts, Susan. “Sloppy software was AIDS disc’s Achilles heal.” New Scientist 125 (6 January 1990): 34.

R.2. National Security Agency

Bamford, V. James. The Puzzle Palace. New York City: Penguin Books, 1983.

Black, Peter. “Soft Kill.” Wired 1 (July/August 1993): 49–50.

“Electric Word.” Wired 1 (September/October 1993): 31.

Holden, Constance, ed. “Viral tall tale?” Science 255 (24 January 1992): 406–7.

Marshall, Eliot. “Worm invades computer networks.” Science 242 (11 November 1988): 855–6.

R.3. Data Encryption

Banerjee, S.K. “High speed implementation of DES” Computers and Security 1 (1982): 261–7.

Brassard, Gilles. Modern Cryptography. New York City: Springer–Verlag, 1988.

Computer Security. Alexandria: Time–Life Books, 1986.

Denning, Dorothy E. R. Cryptography and Data Security. Reading: Addison–Wesley, 1983.

MacMillan, D. “Single chip encrypts data at 14Mb/s” Electronics 54 (16 June 1981): 161–5.

Williams, D. and Hindin, H.J. “Can software do encryption job?” Electronics 53 (3 July 1980): 102–3.

R.4. Virus Creators

Brunner, John. The Shockwave Rider. New York City: Harper & Row, 1975.

Gerrold, David. When Harley Was One. Garden City: Nelson Doubleday, 1972.

Hafner, Katie and Markoff, John. Outlaws and Hackers on the Computer Frontier. New York City: Touchstone, 1991.

Ryan, Thomas J. The Adolescence of P1. New York City: MacMillian, 1977.

Shea, Robert and Wilson, Robert Anton. The Illuminatus Trilogy. New York: Dell, 1975.

R.5. Computer Architecture

Stone, Harold S. and Siewiorek, Daniel P. _Introduction to Computer Organization and Data Structures: PDP–11 Edition_. New York City: McGraw–Hill, 1975.

Tanenbaum, Andrew S. _Structured Computer Organization_. Eaglewood Cliffs: Prentice Hall, 1976.

TMS320C3x Users Guide. Houston: Texas Instruments, 1992.

R.6. Citations

1. “Cornell issues report on computer worm.” Computer 22 (June 1989): 99.
2. Denning, Peter. “The Internet worm.” American Scientist 77 (March/April 1989): 126–8.
3. Eisenberg, Ted. “The Cornell Commission: on Morris and the worm.” CACM 32 (June 1989): 706–9.
4. Fainberg, Tony. “The night the network failed.” New Scientist 121 (4 March 1989): 38–42.
5. Marshall, Eliot. “Worm invades computer networks.” Science 242 (11 November 1988): 855–6.
6. Marshall, Eliot. “The worm’s aftermath.” Science 242 (25 November 1988): 1121–2.
7. Rochlis, Jon. “With microscope and tweezers: the worm from MIT’s perspective.” CACM 32 (June 1989): 689–98.
8. Spafford, Eugene. “Crisis and aftermath (Internet worm).” CACM 32 (June 1989): 678–87.
9. Holden, Constance, ed. “Viral tall tale?” Science 255 (24 January 1992): 406–7.
10. Black, Peter. “Soft Kill.” Wired 1 (July/August 1993): 49–50.
11. “Electric Word.” Wired 1 (September/October 1993): 31.
12. MacMillan, Dave. “Single chip encrypts data at 14Mb/s” Electronics 54 (16 June 1981): 161–5.

13. Williams, Deborah. and Hindin, Harvey J. “Can software do encryption job?” Electronics 53 (3 July 1980): 102–3.
14. Thompson, Ken. “Reflections on trusting trust.” CACM 27 (August 1984): 761-3.
15. Ferbrache, David. A Pathology of Computer Viruses. London: Springer–Verlag, 1992.
16. ---, 31-6.
17. ---, 73-82.
18. Brassard, Gilles. Modern Cryptography. New York City: Springer–Verlag, 1988.
19. Holden, Constance, ed. “Rogue AIDS disk alarms researchers.” Science 247 (5 January 1990): 24.

APPENDIX A - EXOTIC HARDWARE DATASHEETS

A.1. Overview

This appendix contains datasheets for the exotic hardware required for Aardvark. It is not necessarily the case that the vendors specified for these components and subsystems are the only one, but merely those I have used for the basis of the design.

A.2. DE1645EE NVSRAM

A.3. VM007 DES Encrypter/Decrypter

A.4. Touch Memory™

VITAE

Charles James Wilson, B.S.C.S.

PO Box 202287

Austin TX 78720-2287

mailto:pathfinder@acm.org

Professional Objective:

My professional objective is to work for a progressive organization designing and developing the human interface aspects of systems which will allow for less threatening, more productive end–user systems.

The Dreamers Guild, Inc. 6/91 - 8/92

Human Interface Designer / Chief Operating Officer

The Dreamers Guild is a privately held company focused primarily on the entertainment industry working with software and music systems. I consulted on user interface issues of projects being undertaken by the company.

I performed a user interface redesign and restructuring of the underlying data model for a system which interfaced a Hell CMYK color separating scanner to a Macintosh (ScanMac). I also wrote the user documentation for ScanMac.

In February of 1992, I became the company’s chief operating officer responsible for managing the company’s in–progress projects, handling the office equipment management, project timelines and resource allocation.

Ambassador College 8/90 - 5/91

Consultant

Ambassador College is a liberal arts college in Big Sandy, Texas. While there, I lent my skills to their computer services department and advised them on issues of networking, hardware support and equipment acquisition. I specified the requirements for the their computer repair facility and severe electrical storm protection.

Tri–Data Systems, Inc. 4/89 - 8/90

User Interface Designer

Tri–Data Systems, Inc. (later acquired by Avatar) is a telecommunications company which produces microcomputer to IBM mainframe gateways and an IBM 3270 terminal emulator.

I redesigned and re-implemented the company’s IBM 3270 terminal program with special attention being given to the user interface. I added an interface to allow for IBM 3270 function keys to be mapped to the keyboard at the user’s discretion. I also created a multi–programmer/multi–computer development environment platformed on the MPW environment allowing for structured hierarchical projects, programs and the overriding of both libraries and include files.

Additionally, I provided internal Macintosh technical support for MPW, C, hardware and general Macintosh operating system issues. Finally, I provide typographic, layout and copy editing support to the marketing department.

Qubix Graphic Systems, Inc. 5/87 - 4/89

Software Engineer

The Leonardo, is a high–end technical illustration system platformed on Sun Microsystems computers with a proprietary user interface. I created a utility to take very low resolution Kanji raster fonts and produce high resolution vector representations for use with Leonardo. I rewrote the company’s vector font editor to allow for use with the multi–megabyte Kanji fonts. I redesigned the interface of the Leonardo product creating a set of interface guidelines which enabled the streamlining of the product for its use with a standard Sun monitor. I then designed and developed the user interface for the Macintosh version of the Leonardo working with the engineering and marketing departments.

Volt Information Sciences/Autologic 7/85 - 5/87

System Manager/Software Engineer

Volt Information Sciences designed a completely integrated electronic publishing system for the Army. I acted as LAN administrator and systems manager.

Autologic produces a text composition software package called MicroComposer™. I developed MicroComposer’s raster graphics editor. I also developed an implementation of the Kermit protocol for the Convergent Technologies NGEN. Additionally, I provided hardware and software internal technical support.

General Electric Company 3/84 - 6/85

Scientific Applications Programmer

While awaiting my security clearance I worked with Data Systems Resource Management, the research arm of GE Space Systems, where I conducted a study of laser printer usability with the Convergent Technologies NGEN and provided internal CTOS system and Convergent hardware support. I also redesigned and ported a PDL analysis program from the VAX to the Convergent Technologies MegaFrame and NGEN. Following my clearance, I programmed scientific applications in FORTRAN on an IBM 370.

applied computing devices, inc. summer/83

Programmer

I developed the user interface of a telecom system on an IBM Series 1 running under CPIX, a UNIX–like operating system. I also designed and developed a source code management system under the Bourne shell.

ExperCamp summer/81-82

Instructor

Computer Camp, Inc. (now ExperCamp, a division of ExperTelligence, Inc.). was the first commercial computer instruction camp for seven to seventeen year olds. I was both camp counselor and computer language class instructor, teaching BASIC, LOGO, assembler, and FORTH.

Rose–Hulman Institute of Technology 9/80 - 6/84

Grader/Teaching Assistant/System Manager

I graded homework for the FORTRAN, C, and APL language classes. I was also the teaching assistant for the FORTRAN language classes. Additionally, I was the Computer Science Department’s systems manager.