Development of a Programming Model for the AEGIS Secure Processor

by

Ishan Sachdev

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005

Author . . . .
Department of Electrical Engineering and Computer Science
May 19, 2005

Certified by . . . .
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Development of a Programming Model for the AEGIS Secure Processor

by

Ishan Sachdev

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, a high level programming model for the AEGIS secure processor is designed and implemented. The AEGIS processor enables developers to create trusted systems, while only needing to trust the AEGIS processor. In order for developers to utilize the processor, there was a need for high level access to the low level AEGIS constructs. There was also a need for a practical programming model to describe how to correctly design and develop applications utilizing the AEGIS system. The implementation of this programming model was done using a combination of C, Java, and Assembly.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Professor Devadas for giving me the opportunity to work in CSG and on this project. I would like to thank Ed Suh and Charlie O'Donnell for answering my never-ending barrage of questions. Without them, this thesis never would have gotten off of the ground. I would like to thank Russ Cox for being an all-around guru and unravelling the mysteries of GCC for me. Finally, I would like to thank Chris Leung for reminding me that there is such a thing as too much free time.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation for AEGIS Secure Processor

Computing devices are quickly integrating themselves into every facet of our daily lives. The number of interconnections among such devices is also growing at a rapid pace, along with the amount of data being accessed. Furthermore, the popularity of small, wireless devices such as laptops, PDAs, and cellular phones has increased greatly. This has resulted in two new complications. First, as these devices are generally less powerful than necessary, much of the actual computation is no longer being done locally but being farmed out to a remote server. Second, it is now more possible for an adversary to come into physical contact with a trusted device, and therefore tamper with a 'secure' system [12].

These conditions necessitate a secure method of computation that fulfills the following conditions. It must protect against physical tampering which attempts to corrupt data, discover private data, or violate copy protection. It must be able to reliably generate, protect, and share some type of secret which can be used in cryptographic primitives. It should also attempt to reduce the overhead inherent in advanced security measures to only that which is necessary.

The AEGIS single chip secure processor as described in [12] was developed to meet these criteria, and provide a single device on top of which secure systems could be easily constructed. The processor is able to fulfill our security requirements by providing

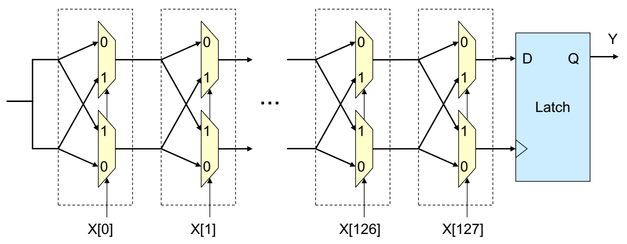


Figure 1-1: Silicon PUF Delay Circuit [12]

a set of secure environments, each offering different levels of security. AEGIS also utilizes Physical Random Functions (PUFs), as described in [5], to veritably create, maintain, and share secrets.

A PUF is a random function that is tied to a physical device in such a way that, given a polynomial amount of physical measurements, it is still extremely difficult to deduce the response to a randomly selected challenge. The PUF utilized in the AEGIS processor makes use of the inherent timing differences in any integrated circuit due to slight variances in the manufacturing process [5]. The delay is measured across a delay circuit like that shown in Figure 1-1. Using these delays, it is possible to differentiate between different PUF-containing integrated circuits. These PUFs have an exponential number of input/output pairs, making model building or full characterization impossible. Thus, only with physical access to the PUF can you hope to arrive at the correct output.

The AEGIS architecture provides four different secure environments, each providing a different level of security: Standard Mode, Tamper-Evident Mode (TE), Private Tamper-Resistant Mode (PTR), and Suspended Secure Processing Mode (SSP). These modes use a combination of Integrity Verification and Memory Encryption to ensure that code and data stays private and untampered with throughout the application's execution. SSP mode allows a developer to temporarily switch out of a secure mode to run untrusted code.

Thus, the AEGIS processor provides a secure, efficient means of developing a secure computational system.

1.2 Motivation For High Level Programming Model

It is not enough to simply provide developers with the AEGIS processor; there is also a need for a high level programming methodology to allow developers to utilize the AEGIS constructs. In particular, there must exist a way for a developer to partition his application code and data among the three different AEGIS secure memory regions, and to partition the execution of his application among the four different security modes. The interaction with AEGIS should be as transparent as possible.

This thesis will demonstrate, using a high level language, such a programming model for creating secure applications. This model separates an application's procedures and data structures into three distinct categories: unprotected, verified, and private, which are then stored in corresponding regions of memory. The model also separates an application's execution into four distinct security modes: Standard, TE, PTR, and SSP.

Unprotected procedures can only run in standard or SSP mode, and are restricted to using only unprotected variables. Verified functions can either run in TE or PTR mode, but in TE mode they are restricted to accessing verified and unprotected variables, while in PTR they can also access private variables. Finally, private procedures run in PTR mode, and can access variables of any type.

In order to utilize the AEGIS system, the developer will be responsible for specifying procedure and data structure memory regions, and switching execution modes appropriately.

1.3 Motivation for Mid-Level Toolchain

To bridge the gap between the low-level AEGIS system calls and a high level programming language such as C, there must exist a library that exposes to developers the low-level functionality. This thesis will describe the implementation of a set of tools that will be incorporated into a library and make the programming model previously discussed possible.

The programming model requires a set of constructs to ensure the security requirements. It requires three different stacks and three different heaps to hold variables of each type. It also requires a segmentation of memory to allow different sections of the application to be stored according to their security parameters.

However, the developer should not have to worry about managing these disparate structures. Their responsibilities should end with typing data correctly and making appropriate security mode switches. Therefore, a set of underlying tools is needed to manage memory at a low level.

These tools will be responsible for: initializing memory regions, making appropriate changes to the underlying stacks on a mode change, managing and retrieving structures allocated on each of the three heaps, and loading the program data and code into the correct memory segments.

There will also need to be methods that allow developers to initialize the AEGIS system, make appropriate AEGIS mode changes, and exit the AEGIS system when security is no longer needed. Developers should not have to interact with the low level assembly commands, and their associated AEGIS parameters and structures. They should be able to fully utilize AEGIS functionality from within the high level language.

1.4 Motivating Example

A particular scenario where the AEGIS system would have a significant impact is sensor networks. Sensor networks are networks of hundreds or thousands of small sensor nodes. These nodes are made up of sensors, some form of actuator for movement, and general purpose computing components. The nodes communicate with each other, and with a central server, wirelessly [6].

Sensor networks are intended to monitor a given environment, such as a battlefield for coordination or forward scouting purposes, or for inventory control in a warehouse. The idea is that through monitoring of a small piece of the environment by a large number of nodes, one can derive a good picture of the total environment.

The primary characteristic of such a sensor network is that the nodes, in order to reduce size, operate on very low power and therefore have low bandwith availability and reduced computational power.

Currently, there are a number of approaches, such as TinySec [6], that examine the problem of sensor net security. However, in general, these solutions only deal with the security of the messages being passed. They do not deal with a number of other important security issues. For example, they do not protect cryptographic material and other secret data contained on the nodes. Using this secret material, an adversary can then place malicious nodes into the network which will will be able to impersonate authorized nodes. These schemes also do not protect against a node performing unauthorized computations in lieu of correct computations.

The AEGIS system can address these security issues, and offer a solution to these problems. Once the AEGIS system has been fully detailed, I will explain its application to sensor networks.

1.5 Organization

This thesis is structured as follows. Chapter 2 discusses related work, including information flow and other secure computing bases. Chapter 3 discusses the AEGIS secure execution modes. Chapter 4 discusses the high level programming model that is exposed to a developer. Chapter 5 discusses the underlying implementation of the programming model. Chapter 6 discusses AEGIS-enabling a sensor network. Finally, Chapter 7 contains some concluding remarks and ideas for future improvements to the programming model.

This work was done in collaboration with G. Edward Suh and Charles W. O'Donnell.

Chapter 2

Related Work

2.1 Trusted Computing

There are a number of platforms that attempt to offer a secure computing environment similar to that of AEGIS. Next Generation Secure Computing Base, Trust Zone, and Trusted Platform Module are three of the more prominent platforms.

Next Generation Secure Computing Base (NGSCB), created by Microsoft, is meant to secure future versions of the Windows operating system. NGSCB offers applications a higher level of security to ensure privacy of data, built in secret keys to perform encryption, and application verification by program hash [7].

Trust Zone, created by ARM, is a secure system integrated with ARM microprocessor cores. It is meant to be a basis for building secure hardware applications. Trust Zone offers applications differing levels of security to ensure data privacy and run trusted code. It also allows for tagging of application data and code, and then appropriate partitioning in physical memory into separate secure and non-secure regions [2].

Trusted Platform Module (TPM), created by the Trusted Computing Group, is a secure computing hardware module meant to be an open industry-wide trusted computing standard. TPM provides internal cryptographic tools and keys, hardware protected storage, and program hashes of applications [3].

All three of these secure platforms share similar features with AEGIS. The main distinction is that AEGIS protects against physical attacks on a computer system, while the other three platforms do not.

Each platform has some form of protected and unprotected memory, data tagging, and secure execution modes. Thus, they all need to have some form of programming model similar to the one outlined in this thesis.

2.2 Information Flow

Information Flow research is concerned with investigating methods of controlling data flow through an application. Myers et al. have done research in this area specifically relating to information flow in secure applications [4, 8, 9]. Other research on information flow has been done by Pottier and Simonet [10] and Abadi [1].

JFlow is the result of Myers' work on statically typing Java applications. JFlow is an extension to the Java language that allows a developer to add statically checked information-flow annotations to application variables. This allows security of information flow to be treated as an extended form of type checking.

The significant contributions of JFlow are as follows. First, it was developed to a be a useable and practical programming model. Previous attempts at such static information flow analysis were too restrictive to be used by developers. Second, JFlow supports a number of language features not previously integrated with static information flow such as subclassing, dynamic type tests, and exceptions. Third, it allows for label polymorphism. This allows for application code to be written independent of the specific label that will exist at run time. Fourth, it supports automatic label inference. This means that labels will be implicitly extended to variables even when there is no specific label assigned to that variable, reducing the work of the developer. Finally, it utilizes the decentralized label model. This model allows multiple principals to ensure their own privacy constraints, even in the presence of other mutually distrusted principals.

JFlow works by allowing a developer to tag each piece of data with a form of security label, describing which principal created the value, and which other principals are allowed to view it. As variables are combined, for example by subtraction of two integer variables, their labels flow to the newly created variable. For example, if principals A and B are allowed to view the variable `secret`, and principals B, C, and D are allowed to the view the variable `notSoSecret`, then only principal B can view the variable `biggerSecret = secret - notSoSecret`. In this way, the security constraints are propagated through the application. Also, individual principals can relax their security constraints if the need arises without affecting the constraints of other principals. For example, in the above situation, if Principal E were allowed to view `secret`, he would still not be allowed to view `notSoSecret` or `biggerSecret`.

Once these type constraint labels have been established by the developer, they can be checked at compile time by the compiler. The compiler can determine if there is any propagation of labels that leads to information leakage. In this way, the information flow through the application can be verified.

Myers and others have extended information flow research to the area of secure program partitioning, of which Jif/split is the result [13, 14]. Secure program partitioning is a technique for protecting private data in an environment with mutually untrusted hosts, such as a distributed computing grid.

Myers' method for secure partitioning involves first annotating application data using a security-typed language, similar to JFlow. Then, based on the security policy created by the security-typed language, an automatic splitter partitions the application's execution across the set of untrusted hosts. In sum, the distributed segments of the computation perform the same computation as the original, unsegmented application. At the same time, the security constraints of each host are satisfied, without requiring a single universally trusted entity.

Currently, the AEGIS programming model does not perform any type of information flow analysis. Ensuring correct information flow of application data is the job of the developer. Also, partitioning the application among the different secure execution modes is also a task that must be performed manually by the developer. Therefore, it would be advantageous to integrate the secure typing and partitioning work described above in order to automate these processes, and provide a more formal method for arriving at a correct AEGIS enabled application.

Chapter 3

Execution Modes

3.1 Overview

The AEGIS processor provides four secure execution modes, each providing an increased level of security for application processes. These four modes are: Standard Mode, Tamper-Evident (TE) Mode, Private Tamper-Resistant (PTR) Mode, and Suspended Secure Processing (SSP) Mode.

Standard mode offers no additional security measures. Tamper-Evident mode verifies the integrity of application state. Private Tamper-Resistant Mode ensures the privacy of application state. Suspended Secure Processing Mode provides an insecure environment with no overhead that limits access to secure application state.

All applications begin in TE mode. From TE mode, switches to the more secure PTR mode and the less secure SSP mode are possible. Symmetrically, from PTR mode, only switches to TE and SSP modes are possible. Once in SSP mode, applications can only run designated functions and then must return to whichever mode SSP was initiated from. On exiting the AEGIS secure system, control is returned to the standard mode.

The AEGIS security mode flow is shown in Figure 3-1.

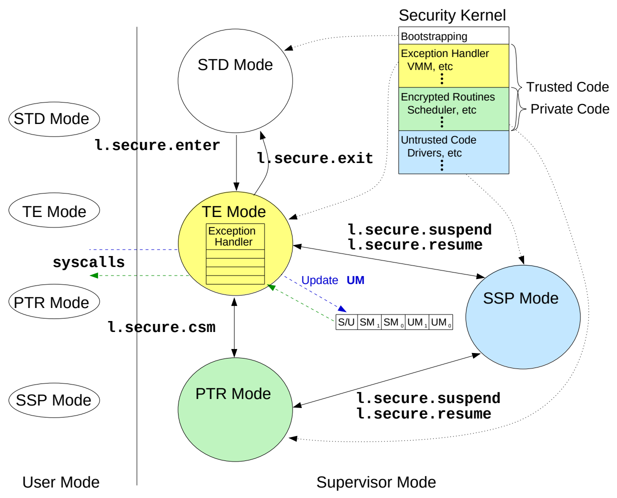


Figure 3-1: AEGIS Secure Modes and Transitions [12]

3.2 Standard Mode

Standard Mode has no additional security features. This is equivalent to how a normal process on a non-AEGIS system would be run. Therefore, Standard Mode has no additional overhead.

3.3 Tamper-Evident Mode

Tamper-Evident (TE) mode ensures the integrity of program state. This ensures that any attempts to tamper with or alter such data will be detected. However, no attempt is made to hide this data from an adversary.

All program state deriving from the TE mode is stored in the Verified region of physical memory. This section of memory is then protected by an integrity verification algorithm, designed to detect any unauthorized changes.

The Verified memory region is further divided into two smaller regions, Static and Dynamic. The Static Verified region is read-only, and therefore should not be changed as the program is running. Therefore, a static cryptographic message authentication code (MAC) is sufficient to protect this sub-section. The MAC need only be computed once at the start of the program's execution since the static data will not change, and therefore the MAC itself should also remain unchanged. The MAC is then stored in a reserved portion of the unprotected memory.

The Dynamic region, however, contains data that will change over the course of a program's execution. This region must therefore be protected against replay attacks. A replay attack is when the new value at a particular memory address is replaced with an older value by an adversary. The processor protects the Dynamic Verified region against this type of attack by using a hash tree.

A hash tree works by creating a tree of hashes over the data to be protected. The root of this hash tree is then stored in a safe location. On each store to memory, the path from the data leaf stored to the root, and therefore the root, is updated. On each load from memory, the path from the data leaf to be loaded is verified.

The AEGIS processor stores this root hash on-chip to ensure that it cannot be tampered with.

3.4 Private Tamper-Resistant Mode

Private Tamper-Resistant (PTR) mode ensures data integrity, as previously described, as well as data privacy. Data privacy is ensured through the use of memory encryption.

All program state derived from PTR mode is stored in the Private memory region. All off-chip data stored in the Private region is encrypted when stored into memory, and decrypted when loaded from memory. Encryption is performed using a One-Time-Pad (OTP) encryption scheme. OTP encryption works by selecting a random sequence of bits (secret key) equivalent in length to the data to be encrypted. This secret key is then combined with the data using the xor function to create the encrypted data. The exact secret key used for encryption is again needed for decryption, and again the xor function is used [12].

As with the Verified region, the Private memory region is further divided into two smaller regions, Static and Dynamic. In general, data contained within the Private Dynamic region will also be protected by the same hash tree that protects the Verified Dynamic data. In certain cases where it is optimal to have a Private memory region without Integrity Verification, time stamps can be maintained for all data in this region to thwart replay attacks. These time stamps will then be protected by the hash tree.

3.5 Suspended Secure Processing Mode

Suspended Secure Processing (SSP) mode is a special, insecure mode used to provide developers with a more flexible security model. SSP mode can only be entered from TE or PTR modes, and must return to the mode it was called from. Code executed while in SSP mode has no security overhead, and also has no access to any of the Verified or Private data being used by the processor.

SSP mode has two major uses. First, developers will almost certainly want to use functionality in large, external existing codebases such as libraries. If forced to run their entire application in one of the secure AEGIS modes, the developer would be responsible for ensuring the security and trust of this external code, a challenging, and ultimately unnecessary or impossible task. By first switching to SSP mode, and then running this code, developers no longer have to worry about the potential for a security breach by utilizing external code.

Second, it is also very likely that an application in its entirety will not need to run in one of the AEGIS secure modes. By utilizing SSP mode, developers can switch out of a secure mode to run code that is not integral to the security of the overall application. By switching to SSP mode, they avoid the computational overhead of the more secure modes. This allows developers the flexibility to optimize their applications for performance.

Chapter 4

High Level Programming Model

4.1 Overview

In order to allow developers to utilize the functionality of the AEGIS processor, the low-level AEGIS instructions must be exposed in a high-level language.

We have developed a programming model which allows developers to utilize the AEGIS secure processor through the high level C programming language [12]. This programming model has the following characteristics:

- Allow developers to store functional code in appropriately secure memory regions
- Allow developers to store data structures in appropriately secure memory regions
- Allow developers to switch execution modes on function calls

4.2 Memory Model

The AEGIS memory model is different from that of a standard processor to accommodate its added security features. Developers have the following view of the physical memory of the system.

There are three basic memory regions: unprotected, verified, and private. Each of these regions is further broken down into four smaller regions: code, data, stack, and heap.

Code and data are placed into the appropriate section at compile time based on how they are declared. Data structures are placed into the appropriate stack based on the AEGIS mode in effect when the data structure is initialized. If declared dynamically, data structures are inserted into the appropriate heap based on how they are created.

The previously discussed verified and private regions encompass these smaller regions. The verified static region includes the verified and private code regions. The private static region includes just the private code region. The verified dynamic region includes the private and verified stack, heap, and data regions. The private dynamic region includes just the private stack, heap, and data regions.

This memory model has two advantages. First, it ensures physical separation of variables with differing security levels. Second, it allows developers the flexibility to decouple functions from AEGIS modes.

Without separate stacks and heaps, physical memory would be made up of blocks of memory regions, each protected at a different security level. It would be very costly and inefficient to segment memory in this fashion. With separate stacks and heaps, AEGIS can be initialized to protect memory regions of known size.

By utilizing separate stacks and heaps, it is also possible to decouple functions from AEGIS modes as local variables will be automatically placed onto the appropriate stack at run-time. Therefore, developers are free to run any function in any mode without worrying about security leakage or improper use concerning local variables.

4.2.1 Data Types

In any application, there are three types of data structures: global structures, structures allocated on the heap, and structures allocated on the stack. A developer has the power to place each of these types of data structures into a different part of memory depending on his security needs.

Stack Variables

Structures allocated on the stack are local variables and data structures declared within the scope of a specific procedure. As far as the developer is concerned, these variables will be placed onto a stack consistent with the current AEGIS mode.

- Standard Mode $\longrightarrow$ Unprotected Stack
- TE Mode $\longrightarrow$ Verified Stack
- PTR Mode $\longrightarrow$ Private Stack

This abstraction provides a number of advantages. First, it ensures that developers do not have to worry about attempting to specify the security level of each local variable when writing an application. Second, it ensures that local variables are not fragmented across physical memory.

The fact that local variables are created in a certain stack based on the current AEGIS mode means that developers do not need to specify the security level of the variable. Forcing developers to assign local variables a security level at development would restrict the utility of each function.

For example, if a particular local variable were declared of type 'private' in Function A, then Function A could only be run in PTR mode. This can cause two types of problems. First, it is not always possible to predict before run time in which AEGIS modes a function might be run. Thus, a developer would need to be certain that

Function A is never run in TE or SSP modes at any point. This could potentially be a very complicated and painstaking process. Second, if the function did in fact need to be run in different AEGIS modes, it might be necessary to have multiple versions of the functions, with local variables declared appropriately. This is unnecessary, and is a problem that is avoided with our model.

By placing all local variables for a given function onto the same stack, they are assured to be contiguous in memory. This means that AEGIS does not have to worry about dealing with defragmented memory, which would quickly degrade performance as utilizing a fine granularity of memory is much more expensive in hardware.

Heap Variables

Heap variables are those that are dynamically allocated at run time. These are variables for which a developer specifically allocates a set amount of memory on the heap, and is then responsible for utilizing. In C, this is achieved through the `malloc()` function and its associated family of memory access functions.

To allow developers maximum flexibility, our programming model has three different heaps as described earlier. Again, this has the advantage of allowing AEGIS to protect memory regions of known size, instead of having to protect memory at word granularity. Access to each heap is governed by a separate `malloc()` function as follows:

- `malloc` $\longrightarrow$ Unprotected Heap
- `malloc_v` $\longrightarrow$ Verified Heap
- `malloc_p` $\longrightarrow$ Private Heap

Each distinct `malloc()` function is used similarly to how the standard C function is used, as shown in Figure 4-1.

- `malloc(size_t size);`
- `malloc_v(size_t size);`
- `malloc_p(size_t size);`

Figure 4-1: AEGIS Malloc Functions

At any particular time during an application's execution, the application can access any heap variables that exist in a heap of equal, or lower, security when compared with the current AEGIS mode.

- Standard Mode ⟶ Unprotected Heap
- TE Mode ⟶ Verified Heap, Unprotected Heap
- PTR Mode ⟶ Private Heap, Verified Heap, Unprotected Heap

Along with `malloc()`, there are two other methods necessary for correct heap management. They are `free()` and `realloc()`. The `free` method is used to deallocate unneeded space on the heap. The `realloc()` method is used to adjust the size of a previously allocated heap variable due to new memory requirements. Each of the previously described `malloc()` functions has an associated `free()` and `realloc()` method as follows:

- `malloc`
  - `free(void *ptr)`
  - `realloc(void *ptr, size_t new_size)`
- `malloc_v`
  - `free_v(void *ptr)`
  - `realloc_v(void *ptr, size_t new_size)`

- `malloc_p`
  - `free_p(void *ptr)`
  - `realloc_p(void *ptr, size_t new_size)`

The usage of these functions is the same as that of their standard C counterparts.

Global Structures

Global structures are variables declared globally that persist through the entire execution of an application. Local static variables are also included in this category since they are treated by the compiler as global variables. Local static variables are only created once and persist across multiple calls to the same function, and therefore are really global variables that are only accessible from within a particular function.

In general, global variables are placed by the compiler into the data memory region of the compiled application. In the AEGIS model, there are three different data memory regions as previously discussed. Developers can place global variables into the correct region by utilizing AEGIS-specific tags as follows:

- `verifieddata` ⟶ Verified Data Region
- `privatedata` ⟶ Private Data Region

Unprotected global variables do not need to be tagged. These tags can be used in an application as in figure 4-2.

4.2.2 Application Code

In general, application code is placed by the compiler into the .text memory region of the compiled application. In the AEGIS model, there are three different memory regions for application code as previously discussed.

```
#include "tags.h"

int x;
long y verifieddata;
char *z privatedata;

void test();

int main()
{
 .
 .
 .
}

void test()
{
 static short test1;
 static int test2 verifieddata;
 static long test3 privatedata;
  .
  .
  .
}
```

Figure 4-2: Global Variable Tagging

Developers can place application code into each section at the function level, depending on their security requirements for each particular function. Functions can be tagged, similarly to global variables, so that the compiler will place them into memory appropriately at compile time as follows:

- `verifiedcode` $\longrightarrow$ Verified Code Region
- `privatecode` $\longrightarrow$ Private Code Region

Unprotected functions do not need to be tagged. These tags can be used in an application as in figure 4-3. Functions must be tagged at the function prototype. Although function prototypes are not required by C, they are required for all functions which a developer wants to tag.

It is anticipated that developers will place most of their code into the Verified Code Region. This ensures that any tampering with application code will be detected. The Private Code Region is reserved for application code that must be kept private, such as that for proprietary algorithms. The Unprotected Region should be used either for hard/impossible to verify code, such as a pre-compiled library, or for code that does not need to be verified for secure application execution.

4.3 Functional Model

There are four different AEGIS modes available to developers, Standard, TE, PTR, and SSP mode. Applications will begin in TE mode, and on exit, return to Standard mode. Therefore, if developers wish to switch to an unsecured mode, they will switch to SSP mode.

The decision was made to begin execution in TE mode because we anticipate that developers will want to execute most of their application in TE mode. Therefore, it makes sense to initialize the application in TE mode.

```
#include "tags.h"

void test1();
long test2() verifiedcode;
char test3() privatecode;

int main()
{
        .
        .
        .

        test1();
        test2();
        test3();

        .
        .
        .
}

void test1()
{
        .
        .
        .
}

long test1()
{
        .
        .
        .
}

char test1()
{
        .
        .
        .
}
```

Figure 4-3: Function Tagging

Developers can switch AEGIS modes only on a function call. Since application developers think in terms of functional units, it seemed that the most appropriate time to make mode changes is on a functional switch. This is also necessary given the memory model that we have selected. Allowing mode changes only on function calls guarantees that all local variables for a given function will be located on the same stack, and ensures the memory constraints described earlier.

Mode changes are made by calling the appropriate AEGIS function, passing it the function call to be executed in the new mode as an argument. The mode-change functions are as follows:

- `S_SSP([function call])` ⟶ Suspended Secure Processing Mode
- `S_TE([function call])` ⟶ Tamper-Evident Mode
- `S_PTR([function call])` ⟶ Private Tamper-Resistant Mode

Developers must call these mode-change functions in one of the following two fashions:

- No Return Value ⟶ `S_SSP([function call])`
- Return Value ⟶ `S_SSP([returnVal]=[function call])`

This means that function calls that involve AEGIS mode changes cannot be used as a part of a larger expression, such as an equation, or as a parameter in another function call.

Only allowing functional calls of this format was an implementation decision, and will be discussed further in Section 5.5.

These functions are used as in figure 4-4.

Developers do not need to switch to Standard Mode, or exit the AEGIS system, as this is done automatically at the end of an application. There is no need to exit

```
int summer(int w, int x, int y, int z) verifiedcode;
int verify(int test) verifiedcode;
void transmit(int data);

int main()
{
        .
        .
        .
        S_TE(z=summer(2,3,4,5););

        S_SSP(transmit(z););
        return 0;
}

int summer(int w, int x, int y, int z)
{
        int sum=w+x+y+z;
        int result=0;

        S_PTR(result=verify(sum););

        return result;
}

int verify(int test)
{
        int key=14;
        return(test==key);
}
```

Figure 4-4: AEGIS Mode Changes

AEGIS during an application's execution as switching to SSP mode accomplishes the same goal as exiting AEGIS and switching to Standard Mode.

However, in the case of an unexpected execution break, such as `exit(1)`, proper security protocol must still be followed to ensure the integrity of the AEGIS system. Therefore, the standard C `exit()` method has been replaced with an AEGIS-specific version. The syntax remains the same. This `exit()` method clears all potentially sensitive memory, and then halts the application.

- `exit(int status)`

Chapter 5

Implementation

5.1 Overview

The AEGIS toolchain was implemented in a combination of C, Java, and Assembly. There are three main aspects of the AEGIS implementation: tools for initializing the AEGIS system, tools for making AEGIS mode changes, and tools for exiting the AEGIS system. There are also a number of additional tools required to utilize the AEGIS memory system. Section 5.3 discusses the implementation of AEGIS initialization. Section 5.4 and Section 5.5 discuss the implementation of AEGIS mode changes. Section 5.6 discusses the implementation of exiting the AEGIS system. Finally, Section 5.7 discusses the implementation of additional AEGIS tools.

5.2 Assumptions

The following implementation makes two important assumptions. First, there is no operating system and therefore no virtual memory system. AEGIS is designed to work with an operating system and virtual memory manager, but as yet this has not been implemented.

Second, in our implementation we made an effort to avoid modifying the GCC calling convention and front-end. It is possible to obtain the same functionality described previously by modifiying GCC appropriately. However, this is a very messy and complicated process which we decided to avoid. As we will discuss later, this is an area for future research.

5.3 AEGIS Initialization

There are three main steps to initialize the AEGIS system, culminating in executing the `l.aegis.enter` instruction.

1. Set up physical memory
2. Set up AEGIS entry parameters
3. Enter AEGIS

There are three code segments involved in AEGIS initialization: the AEGIS linker script, the AEGIS bootstrapper, and the enter_aegis() method. The linker script instructs the linker on how to create the object file. The bootstrapper is the first code run by the AEGIS processor on application execution. Finally, `enter_aegis()` is the function responsible for preparing for, and calling, the `l.aegis.enter` instruction.

Section 5.3.1 will discuss initializing physical memory. Section 5.3.2 will discuss setting up the AEGIS entry parameters. Finally, Section 5.3.3 will discuss how to correctly make the call `l.aegis.enter`.

5.3.1 Physical Memory

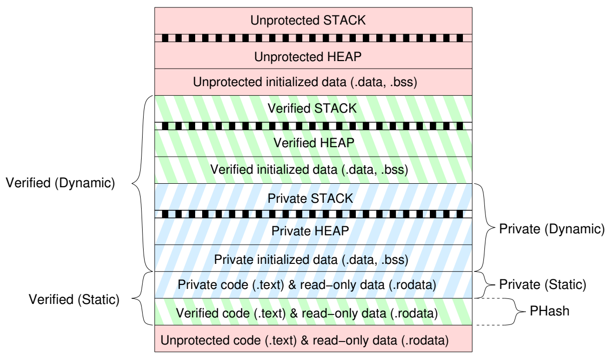
When linking an application, physical memory must be correctly partitioned according to the AEGIS memory model. This is done by the AEGIS linker script.

The purpose of the linker script is to partition physical memory and initialize any linker symbols. The AEGIS linker script partitions memory into the regions described in Section 4.2. There are four regions for each security level as follows: code, data, heap and stack.

1. Unprotected Region
    - Unprotected Code $\longrightarrow$ ucode
    - Unprotected Data $\longrightarrow$ udata
    - Unprotected Heap $\longrightarrow$ uheap
    - Unprotected Stack $\longrightarrow$ ustack
2. Verified Region
    - Verified Code $\longrightarrow$ vcode
    - Verified Data $\longrightarrow$ vdata
    - Verified Heap $\longrightarrow$ vheap
    - Verified Stack $\longrightarrow$ vstack
3. Private Region
    - Private Code $\longrightarrow$ pcode
    - Private Data $\longrightarrow$ pdata
    - Private Heap $\longrightarrow$ pheap
    - Private Stack $\longrightarrow$ pstack

The linker will populate the code and data regions with the appropriate application code and data, and allocate memory for each stack and heap region. The layout of physical memory is shown in Figure 5-1.

Application code and read only data are placed into the correct code sections based on their AEGIS tags as described in Section 4.2.2. Application data such as C variables is placed into the appropriate data region based on its AEGIS tag as described in Section 4.2.1. All untagged code and data is placed into the appropriate Unprotected region. Finally, the .bss section, which contains reserved memory for uninitialized global and static variables, is placed in the Unprotected region. As all global and static variables meant to be protected will be tagged as such, only



Unprotected STACK
Unprotected HEAP
Unprotected initialized data (.data, .bss)
Verified STACK
Verified HEAP
Verified initialized data (.data, .bss)
Private STACK
Private HEAP
Private initialized data (.data, .bss)
Private code (.text) & read–only data (.rodata)
Verified code (.text) & read–only data (.rodata)
Unprotected code (.text) & read–only data (.rodata)
Verified (Dynamic)
Verified (Static)
Private (Dynamic)
Private (Static)
PHash

Figure 5-1: AEGIS Physical Memory Layout [12]

unprotected global and static variables will be marked as .bss. Thus, this section can be included in the Unprotected region.

Each heap and stack is allocated a set amount of memory. These values can in fact be set before compilation to ensure optimum memory usage based on the particular application being run.

Stacks

The three stack pointers must also be properly initialized. The C compiler, however, only knows of the existence of one stack. Thus, the stack pointer for the current stack is loaded into r1 (stack pointer register), and the remaining two stack pointers are stored in memory. The memory locations for these three stack pointers are determined by the linker in the linker script, as are the initial values for the stack pointers themselves.

Next, at beginning of the AEGIS bootstrapper, the verified stack pointer is switched into r1.

Finally, in the `enter_aegis()` function, the initial locations of the three stack pointers, set up by the linker, are stored to their preset memory locations.

5.3.2 AEGIS Entry Parameters

When AEGIS is initialized, it must be told how physical memory is partitioned so that it knows which regions to protect in which manner. This is accomplished by setting up a table in memory describing the necessary regions, and then passing this information to AEGIS when entered.

The required regions, and their inclusive memory regions, are shown in Figure 5-2.

Verified Static	vcode, pcode
Verified Dynamic	vdata, pdata, vstack, vheap, pstack, pheap
Private Static	pcode
Private Dynamic	pdata, pheap, pstack
Program Hash	vcode

Figure 5-2: AEGIS Memory Regions

The first four regions are as previously discussed. The final region, the Program Hash, defines the region containing the application code and read only data over which AEGIS should calculate the necessary program hash. This program hash is then compared to an existing hash to ensure that the correct application is being executed.

These AEGIS parameters are set up in two steps. First, in the linker script, the required linker symbols are initialized. These symbols are initialized at link time, and are then available to the running C application to use as `extern` variables.

The C function `enter_aegis()`, contains the second part of the AEGIS initialization code. This function utilizes the linker symbols to create the table required by `l.aegis.enter`, and store it in memory.

5.3.3 Enter AEGIS

The final step in initializing the AEGIS system is to call the l.aegis.enter instruction. Once this call is made, however, two checks must be performed before application execution can continue. If either of these checks fail, AEGIS is immediately exited. These checks, and the initial call to `l.aegis.enter`, take place in the `enter_aegis()` function.

First, AEGIS must check that the l.aegis.enter instruction is in the correct location in the executable. This arises from the fact that the Program Hash region is calculated using an offset, calculated at compile time, from the l.aegis.enter instruction. If this check is not made, an adversary could shift all code in the object file by some amount. This could potentially cause two problems. First, if the application assembly contains absolute jumps, i.e. jumps to a specific location in memory, they will now jump to an incorrect instruction, breaking the security model. Second, the program hash, when taken, would only include trusted code, but the Program Hash region itself would include a different set of data due to the fact that the object file had been modified. Again, this would break the security model.

This check is accomplished by setting an assembly-level label at the l.aegis.enter instruction. Then, at execution, the location of this label is compared with the location of the l.aegis.enter instruction set at compile-time as in Figure 5-3. The label `enter_loc:` marks the `l.aegis.enter` instruction. The last three Assembly commands retrieve the current Program Counter during execution, and check that the Program Counter at line 1 is one word away from the `enter_loc label`.

Second, AEGIS must check a hash of the private code and data regions. As previously discussed, AEGIS utilizes a hash of the application to ensure that the correct application is being executed. However, private code and data are stored encrypted in memory. Thus, before actually executing application code, this private

```
    asm volatile("l.add r27,r0,%0" :: "r" (enter_aegis_loc));
    asm volatile("enter_loc:\n"
             "l.aegis.enter r31,r29\n"
             "l.mfspr r25,r0,0x0010\n"
             "l.addi r27,r27,0x4\n"
             "l.sfne r25,r27\n"
             "l.bf exit_aegis\n");
```

Figure 5-3: Verify Location of l.aegis.enter Instruction

region also be verified. The private region is verified by first entering PTR mode. Once in PTR mode, the encrypted private code and data are decrypted by AEGIS utilizing the application provided secret key. A hash of the pcode and pdata sections is then created using the SHA-1 algorithm. This hash is compared with the correct hash of the private code and data region, stored at a known location in Verified memory, where it is also protected and included in the program hash.

Finally, once both checks have passed, the AEGIS system is in effect.

5.4 AEGIS Mode Change Mechanics

As previously discussed, AEGIS mode changes occur only on function calls. Mode changes have the following syntax:

- No Return Value ⟶ `S_SSP([function call])`
- Return Value ⟶ `S_SSP([returnVal]=[function call])`

The AEGIS mode before the function call is called the caller-mode. The AEGIS mode being switched to, in this particular case SSP mode, is called the callee-mode.

Actual execution flows around a mode change is as follows:

Caller-function → Mode Change (to callee-mode) → Callee-function → Mode Change (to caller-mode) → Caller Function

In other words, on a function call involving a mode change, the mode is changed to the callee-mode, the callee-function is run, and then the mode is changed back to the caller-mode and control is returned to the caller-function. This removes the need for the developer to revert to the callee-mode, and ensures that mode changes equate exactly with functional calls.

Switching to Callee-Mode

When making the initial switch to callee-mode, there are six tasks that must be performed prior to making the actual AEGIS switch.

1. Transfer required variables to new stack
2. Save Current Stack Pointer
3. Store all GPRs
4. Clear all GPRs
5. Load new stack pointer
6. Switch to new mode

When making a function call, the calling function can pass parameters to the callee function. These parameters are generally passed on the stack to the callee function. However, AEGIS uses three different stacks. Therefore, after a mode switch, AEGIS will not be able to locate the required parameters on the caller-stack. So, prior to the actual function call, these variables, or at least their values, must be moved to the callee stack.

Local variables are located by GCC through an offset from the stack pointer. For example, let us take a local C variable `int counter`. This variable is referenced by GCC as an offset in bytes from the location pointed to by the stack pointer. For example, `int counter` might be 16 bytes (4 words) offset from the stack pointer. Even after the caller stack has been switched to the callee stack, GCC will still believe

```
//transfer variable temp2 to new stack

//find absolute location of temp2 in caller stack
asm volatile("l.add r31,%0,r0" :: "r" (&temp2));
asm volatile("l.lwz r29,0x0(r31)");

//calculate offset from stack pointer
int _currstack1=0;
asm volatile("l.add %0,r1,r0" : "=r" (_currstack1) :);
int _offset1=(int)&temp2 - (int)_currstack1;

//store the value of temp2 into callee stack
asm volatile("l.add r23,%0,%1" :: "r" (_offset1), "r" (vstack_end));
asm volatile("l.sw 0x0(r23),r29");
```

Figure 5-4: Transferring Variable From Caller Stack to Callee Stack

that `int counter` is located 16 bytes away from the new stack pointer. Thus, if the value of `int counter` is copied to a location 16 bytes away from the callee stack pointer, GCC can still utilize `int counter` as if it were a local variable on the new stack. Our method of copying parameters was therefore to copy their values to a location in the callee stack with the same offset as the parameter's original location in the caller stack, as shown in Figure 5-4.

However, this solution still has one problem. It is possible that a parameter in the function call might not be a local variable, but a variable in another stack frame, and is still being referenced by looking in that stack frame. For example, if an array has been passed into the caller function, and a parameter to the callee function is an element of that array, that element does not exist in the local stack frame as a local variable. It exists as an element of an array in another stack frame, pointed to by the array pointer passed into the caller function.

To avoid this problem, a new temporary variable `temp1, temp2, ... tempn` is created for each parameter in the caller stack. These temporary variables are created as type void*, since the true type of each parameter is unknown. Each

```
//original variables
int* weight = malloc(sizeof(int));
*x = 4;
int volume=6;

//original function call
z=summer(2,3,*weight,volume);

                         ↓

//new temporary variables
void* temp1=2;
void* temp2=3;
void* temp3=*weight;
void* temp4=volume;

//rewritten function call
z=summer(temp1,temp2,temp3,temp4);
```

Figure 5-5: Transforming Function Call

temporary variable is then assigned the value of its corresponding parameter. Since every parameter must evaluate to a value of some kind, these values now exist in local variables on the current stack frame. The function call is then rewritten using the temporary variables as parameters.

In order to ensure that the correct stack is used, r1 must be replaced with the appropriate stack pointer. However, before this is done, AEGIS must save the value of the caller stack pointer so that it can be correctly retrieved when necessary.

When switching from a higher-security mode, such as PTR mode, to a lower-security mode, such as TE mode, there is the potential for information leakage through the General Purpose Registers (GRPs). If they were used by the PTR function to temporarily hold private data, this data would still be in the GPRs when the TE function was called and run. To ensure that this situation does not occur, the GPRs need to be cleared out before the TE function is called. However, the compiler assumes that these values are the same before and after the function call, so AEGIS must therefore restore the correct values post function call. The solution is to store the GPR values on the caller-stack and then zero the GPRs before calling the function.

Finally, the switch to the callee-mode must be made. This is done by the `l.aegis.cam` instruction.

Returning to Caller-Mode

When returning to caller-mode, there are five tasks that must be performed.

1. Switch back to caller-mode
2. Save callee-mode stack pointer
3. Restore caller-mode stack pointer
4. Restore GPRs
5. Place return value into correct variable

Just as when switching to the callee-mode, when switching back to the caller-mode, the callee-mode stack pointer must be stored to memory and the caller-mode stack pointer loaded into r1.

The GPRs that were stored to memory before the function call must also be restored for correct application execution. Then, the mode can be changed back to the caller-mode through the `l.aegis.cam` instruction.

The last task is to place the return value into the correct return variable. This needs to be done because the return variable exists on the caller-stack, while the return value exists only in the return value register, r11. Therefore, only once the stack pointer has been changed back to the caller stack pointer, the value in r11 can be placed into the correct return value. However, if the function does not have a return value, r11 will be set to 0 in order to prevent information leakage.

5.5 AEGIS Mode Change Implementation

The primary decision concerning the implementation of AEGIS mode changes was what form the `S_SSP()`, `S_TE()`, and `S_PTR()` 'functions' should take. Should they be C functions, C defines, or some form of pre-processing tag? The decision was made to implement these 'functions' as pre-processing tags.

This means that once a developer has created an application file, they must run it through the AEGIS pre-processor. This pre-processor was written in JAVA, and searches for the AEGIS mode-change tags `S_SSP()`, `S_TE()`, and `S_PTR()`. Upon finding a tag, the pre-processor replaces the tagged function call with the necessary C and Assembly code to perform the tasks described in the previous section.

The decision to utilize a pre-processor instead of an actual C function or C define for AEGIS mode changes was made for a few reasons. The central problem with using a C function to make the mode change is that the C function would need to generate, on the fly, a new function call using the previously discussed temporary parameter variables. This would mean creating a function call to an unknown function with an unknown number of parameters, which is impossible in C. It is possible, using a function pointer, to call an unknown function. However, the definition of such a call must include a specific number of arguments. Therefore, this was not a viable option. In order to create the correct functional call, a pre-processor was needed that could create the call ahead of time, and was not constrained by the C run-time functionality.

5.6 AEGIS Exit

There are two code segments involved in exiting AEGIS: the AEGIS bootstrapper and the `exit()` method. `exit()` is responsible for preparing for, and finally exiting, the AEGIS system. Our `exit()` method replaces the standard C `exit()` method. The AEGIS bootstrapper is responsible for invoking `exit()`.

When exiting the AEGIS system, the primary concern is to ensure that all private data is wiped from memory to ensure no leakage. Private data can persist in two places: AEGIS registers and system memory. The memory that we are concerned about is only memory that could contain private data: private heap and private stack. The actual heap and stack are encrypted, however data from the heap or stack sitting in the data cache or instruction cache will not be encrypted. Therefore, on exit, these three locations must be cleared of all data. This can be done by loading a 0 into the registers, storing a 0 to all memory locations in the private heap and private stack, and invalidating the instruction cache. Then, AEGIS switches back to the unprotected stack and calls the `l.aegis.exit` instruction to exit the AEGIS system. This functionality is all contained within the `exit()` method.

5.7 AEGIS Tools

In order to utilize the AEGIS system, two additional tools are necessary. First, a method for tagging functions and variables as verified and private. Second, a tool for encrypting private application code prior to application execution. Third, three distinct `malloc()` functions for allocating variables on each of the three different heaps.

5.7.1 AEGIS Tags

As discussed in Section 4.2.1 and Section 4.2.2, there are four AEGIS tags for tagging data and code so that it is placed into the correct Verified or Protected region.

- verifiedcode
- verifieddata
- privatecode

- privatedata

These tags are created using C attributes. C attributes allow a developer to specify into which memory region application code or data will be placed when the application is compiled into a C object file. By defining the above tags as specific C attributes, these tags can be used to place code and data into the correct section of the object file. This is shown in Figure 5-6.

```
#define verifiedcode __attribute__((section(".verifiedcode")))
#define privatecode __attribute__((section(".privatecode")))
#define verifieddata __attribute__((section(".verifiedddata")))
#define privatedata __attribute__((section(".privatedata")))
```

Figure 5-6: C Attributes

5.7.2 AEGIS Private Code Encryption

In order to ensure the privacy of code flagged as private, the Private Code memory region must be encrypted prior to an application's execution. This is accomplished by editing the final, compiled binary executable and encrypting the appropriate sections using the AEGIS encryption pre-processor.

This pre-processor is written in C. It takes as input the compiled AEGIS executable, and outputs an AEGIS executable with the appropriate Private Region encrypted.

The pre-processor works as follows. As AEGIS executables are in the ELF (Executable and Linking Format) binary format, section information can be retrieved utilizing information stored in the executable.

The main task is to locate the Private Code region of the file. For each section, there is a section header which contains, among other information, the location of the actual section data within the ELF file. Section names are contained within a String Table, which is contained in its own section of the ELF file. Section headers

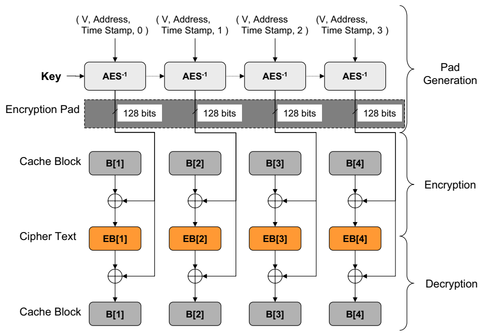


Figure 5-7: AEGIS OTP Pad Encryption of Private Code [11]

themselves do not contain the actual name of the section, but an index into this String Table where the name is stored. So, by iterating through all the section headers and checking the corresponding location in the String Table, .pcode section header can be identified. From this header, the location of the .pcode instructions can be found in the ELF file. The .pcode section is encrypted at a cache block granularity (64 bytes), since this is how instructions will be decrypted by the AEGIS processor.

Encryption of this section is a two-step process. First, a One Time Pad (OTP) is generated for each cache block sized segment. The OTP is generated by using AES encryption to create a OTP specific to the application. As AES creates 128 bit outputs, it can only generate a 128 bit pad. Thus, for each 512 bit cache block, four AES encryptions are needed. The inputs to AES are a bit vector V (the program hash), the address of the cache block being encrypted, a time stamp, and a number denoting the segment of the current cache block being encrypted. The AES key is provided by the developer.

Second, the appropriate segment of .pcode is combined with the OTP using xor to created the encrypted version instruction set. The .pcode section of the ELF binary is then replaced with the new encrypted version. The entire process is shown in Figure 5-7.

5.7.3 AEGIS Malloc Functions

In order to take advantage of the three heaps available in the AEGIS system, there must also be three heap allocation functions. In C, the general heap allocation function is `malloc()`. Therefore, three different versions of `malloc()` were created: `malloc()`, `malloc_v()`, and `malloc_p()`. Symmetrically, three different versions of `free()` and `realloc()` were created: `free()`, `free_v()`, `free_p()`, `realloc()`, `realloc_v()`, and `realloc_p()`.

- `malloc`, `free`, `realloc` $\longrightarrow$ Unprotected Heap
- `malloc_v`, `free_v`, `realloc_v` $\longrightarrow$ Verified Heap
- `malloc_p`, `free_p`, `realloc_p` $\longrightarrow$ Private Heap

The general `malloc()`, `free()`, and `realloc()` functions are simply the ones that exist in the C standard library. The other functions were created by taking the general `malloc()`, `free()`, and `realloc()` implementations, and changing the variable defining where the heap begins and ends.

The AEGIS linker script defines three different heap symbols: `end`, `__AEGIS__end_v`, and `__AEGIS__end_p`. Each symbol defines the end of its corresponding heap, `end` being the symbol used by the C standard library. Thus, three different functions can be created by utilizing the correct heap symbol.

The heap symbol is utilized by `malloc()` when it actually obtains physical memory for a given variable. This is done by the `sbrk()` function. So, two additional versions of `malloc()` were created by creating two additional versions of `sbrk()`, each utilizing a different heap end symbol. Similarly, three different versions of `free()` and `realloc()` were created. When each function is used, the system will allocate memory into the correct heap section as described by the linker script, ensuring that each heap is correctly protected by AEGIS.

Chapter 6

Application to Sensor Networks

Now that the AEGIS system has been fully explained, we will revisit the topic of sensor networks, and explain the application of AEGIS to these networks.

As described earlier, the security problems faced by sensor networks are ensuring the secrecy of cryptographic material and ensuring accurate and authorized computation.

By writing all applications running on the sensor network nodes using AEGIS, these problems can be alleviated. As an example, let us take a simple data retrieval application. This application retrieves information from the environment, performs a computation on it, and then transmits the result of the computation to a central server. More specifically, the application retrieves data from its external sensors. A computation is then performed on the data. The node then generates a secret key. Utilizing this secret key and the computed data, a MACed message is created. The node then transmits this message to the central server.

The relevant section of a possible data retrieval application, `nodeFunction`, is shown in Figure 6-1.

There are now two steps to apply AEGIS to this application. First, global data and application functions must be partitioned in physical memory. Second, application

```
long secretKey;

long generateSecretKey();
long retrieveData();
void transmitData(long data);
long performComputation(long data);
long mac(long clearText);

void nodeFunction()
{
    long retrievedData;
    long computedData;
    long message;

    retrievedData = retrieveData();

    computedData = performComputation(retrievedData);

    message = mac(computedData);

    transmitData(message);
}


void mac(long clearText)
{
    secretKey = generateSecretKey();
    .
    .
    .
}
```

Figure 6-1: Sensor Network Code

execution must be partitioned across the different AEGIS security modes.

First, let us partition data and functional code.

The secret key is a cryptographic value which needs to be kept private. Therefore, it is tagged as `privatedata` and will be placed into the Private region of memory.

The retrieved data, the computed data, and the final message are local variables in `nodeFunction()`. Therefore, since `nodeFunction()` executes in TE mode, these variables will be created on the Verified stack.

All the functions, with the exception of `retrieveData()` and `transmitData()`, will be placed into the Verified region of memory to ensure they are not tampered with. Since the `retrieveData()` and `transmitData()` functions do need to be verified for secure execution, they will be placed into the Unprotected region of memory.

Second, let us partition the execution of the application across the AEGIS security modes. Since the `retrieveData()` and `transmitData()` functions are I/O functions, they can be run in SSP mode. We have assumed that an adversary can modify the data sent over the wireless network, and therefore there is no reason to run these functions in a more secure mode. Since `mac()` uses private data (the secret key), in must be run in PTR mode. Finally, since `performComputation` is important to correct application execution, but does not utilize private data, it should be run in TE mode.

The relevant section of the AEGIS-enabled version of the application is shown in Figure 6-2.

Now, the application has been secured against the previously discussed issues. All private data will be kept private. We can be sure that the node is executing the version of the retrieval function that it is supposed to execute. Finally, the node can authenticate itself to the server verifiably. This would not be possible without the AEGIS system.

Furthermore, from this application the usefulness and necessity of a high level

```
long secretKey privatedata;

long generateSecretKey() verifiedcode;
long retrieveData();
void transmitData(long data);
long performComputation(long data) verifiedcode;
long mac(long clearText) verifiedcode;

void nodeFunction()
{
        long retrievedData;
        long computedData;
        long message;

    S_SSP(retrievedData=retrieveData(););

    computedData = performComputation(retrievedData);

    S_PTR(message=mac(computedData););

    S_SSP(transmitData(message););
}


void mac(long clearText)
{
    secretKey = generateSecretKey();
    .
    .
    .
}
```

Figure 6-2: AEGIS Enabled Sensor Network Code

programming model is clear. Without such a model, enabling such an application for AEGIS would mean writing all the requisite mode changes, memory partitioning, and AEGIS initialization/exit code in assembly. This is not a simple process, and requires a complete low-level understanding of the AEGIS system. It is not realistic to expect that any developer would go through such a process to use the system. However, with the high level model described in this thesis, it is extremely easy to AEGIS-enable the application, and also very transparent to the developer. Thus, the utility of the high-level programming model can be seen.

Chapter 7

Conclusion

In this thesis, we have presented a high level programming model for the AEGIS processor. This programming model exposes the low level AEGIS instruction set to application developers in the high level C language, and allows them to almost transparently utilize the AEGIS system.

The principal parts of the implementation concerned AEGIS initialization, AEGIS mode changes, and exiting AEGIS. There was also an additional set of tools necessary in order to correctly partition application code and data in physical memory.

Finally, we saw how to apply this programming model in the sensor net scenario. This example showed how to AEGIS-enable an application, and also displayed the usefulness and necessity of a high level programming model.

7.1 Future Work

This programming model should be regarded as a starting point for developing a complete, integrated programming model for a secure processor. There is still a significant amount of future research that can be conducted in this area.

First, the current implementation works on top of the GCC calling convention and the C language. It might be possible, and more useful, to integrate the programming model more tightly with the compiler and with the programming language. This would mean the compiler would be aware of the AEGIS security model including the different heaps, stacks, and variable types. Close integration would allow developers more flexibility in using AEGIS combined with a high level programming language, and reduce the number of syntax restrictions. More importantly, by making the compiler aware of the security systems at work, it will be possible to run compile-time checks on the security of an application. Developers would be able to determine, at compile time, if there were any potential security leaks in their application.

Second, it was beyond the scope of this thesis to attempt to determine an optimal method for partitioning application code and data between the three security levels available. Since integrity verification and encryption have a significant overhead, the more application code executed in TE and PTR modes, the higher the added overhead. Thus, ideally one would like to execute as little code as possible in the more secure modes, while still fulfilling the security requirements of the overall application. How to do this optimally is still an open question.

Third, a more formal calculation of the performance overhead of the AEGIS system utilizing this programming model could be performed. This would aid developers in determining what kind of performance hit they could expect, and how they could best optimize their applications to avoid as much overhead as possible.

Finally, currently there is no support through this model for interacting with an operating system and virtual memory system. These features are not currently available on the existing AEGIS implementation. However, AEGIS is intended for use in a multi-user, operating system environment with its associated virtual memory manager. In order to utilize this memory model with such a system, additional work will have to be done to modify the AEGIS `malloc()` methods.

Bibliography

[1] Martín Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, 1999.

[2] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. ARM White Paper, July, 2004.

[3] Sundeep Bajikar. Trusted platform module (tpm) based security on notebook pcs. Intel White Paper, June 20, 2002.

[4] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 198–209, New York, NY, USA, 2004. ACM Press.

[5] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Controlled physical random functions. In *Proceedings of the 18th Annual Computer Security Applications Conference*, December, 2002.

[6] Chris Karlof, Naveen Sastry, and David Wagner. Tinysec: a link layer security architecture for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 162–175, New York, NY, USA, 2004. ACM Press.

[7] Microsoft. Microsoft Next-Generation Secure Computing Base - Technical FAQ. http://www.microsoft.com/technet/archive/security/news/ngscb.mspx, July, 2003.

[8] Andrew C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM Press.

[9] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

[10] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, 2003.

[11] G. Edward Suh, Charles W. O'Donnell, and Srinivas Devadas. Aegis: A single-chip secure processor. In *Proceedings of the 32nd International Symposium on Computer Architecture*, June, 2005.

[12] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of the AEGIS single-chip secure processor using physical random functions. In *Proceedings of the 32nd International Symposium on Computer Architecture*, June, 2005.

[13] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: secure program partitioning. In *SOSP '01: Proceedings of the 18th ACM symposium on Operating systems principles*, pages 1–14, New York, NY, USA, 2001. ACM Press.

[14] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3):283–328, 2002.