Secure Application Partitioning for Intellectual Property Protection

by

Charles W. O'Donnell

B.S. in Computer Engineering, Columbia University in the City of New York, 2003

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

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2005

Author . . . .
Department of Electrical Engineering and Computer Science
August 26, 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

Secure Application Partitioning for Intellectual Property Protection

by

Charles W. O'Donnell

Submitted to the Department of Electrical Engineering and Computer Science
on August 26, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Intellectual property protection is a major concern for both hardware and software architects today. Recently secure platforms have been proposed to protect the privacy of application code and enforce that an application can only be run or accessed by authorized hosts. Unfortunately, these capabilities incur a sizeable performance overhead. Partitioning an application into secure and insecure regions can help diminish overheads but invalidates guarantees of privacy and access control.

This work examines the problem of securely partitioning an application into public and private regions so that private code confidentiality is guaranteed and only authorized hosts can execute the application. This problem must be framed within the context of whole application execution for any solution to have meaning, which is a critical point when evaluating software security. The adversarial model presented balances practical generality with concrete security guarantees, and it is shown that under this model the best attack possible is a "Memoization Attack." A practical Memoization Attack is implemented, and experimentation reveals that naive partitioning strategies can expose the functionality of hidden code in real applications, allowing unauthorized execution. To protect against such an attack, a set of indicators are presented that enable an application designer to identify these insecure application code regions. Finally, a partitioning methodology is discussed that uses these indicators to partition an application in a manner that protects the privacy of intellectual property and prohibits unauthorized execution.

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

Acknowledgements

I would like to thank Srini for encouraging me to pursue whatever ideas I find interesting, for his inexhaustible energy, and for his suggestions which led to this thesis topic. I would like to thank Ed for being an outstanding mentor and for his considerable help in much of my work, including help in giving this thesis direction. I would like to thank Marten for suffering through many hours of theoretical musings concerning the nature of this unwieldy problem. I would also like to thank Dan and Blaise for their useful comments during the writing of this text. I would especially like to thank all of my labmates who consistently remind me that its my peers who teach me the most, including Ed, Daihyun, Prahbat, Ian, Nirav, Michael, Karen, Dan, Dave, Dwaine, Blaise, Jae, Albert, Bill, Chris, Mike, and Rob. I would love to thank Adrienne for her fine editorial skills, and for keeping me company at the beach and ever since. Finally, I would like to thank my parents and family, who played the greatest role in my accomplishments (and existence).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the increasing interconnectivity of computing devices worldwide, system and application security has become a first order concern for both hardware and software architects. One chief worry for these designers is the guarantee that their product will be used in its original and intended fashion.

This concern can be broken into three major considerations. First, software inventors may want to protect the intellectual effort that went into the making of an application. That is, the secrecy of their Intellectual Property (IP) must be maintained even after an application has been distributed throughout the world. Second, designers commonly desire control over who can use their application and who cannot. This is broadly referred to as software piracy prevention or software licensing. Third, benevolent inventors do not want their application to be used maliciously, which may decrease the productivity of their application and others. The ability of an application to withstand malicious attacks which intend to modify the functionality of software is often referred to as application tamper-resistance.

These desires stem from a number of motives, including a designer's altruistic intentions, as well as his capitalistic disposition and financial dependencies. Software businesses must adhere to economic models that allow developers to actually get paid for their work. More often than not, this requires individuals or organizations to purchase the rights to use an application. Without control over who can use an application and who cannot, there is no consumer incentive to purchase the rights to use an application. Similarly, without the protection of intellectual property, organizations can circumvent the required purchase by developing their own copy of an application based on IP found in the original application. Finally, preventing malicious attack is not only a generally agreed "good idea," but also saves businesses large amounts of money that can be lost through patch distribution and decline in consumer confidence.

1.1 Existing Solutions

Although countless techniques and architectures have been proposed to tackle one or all of these points we find it prudent to only focus on physically secure solutions. These solutions make software protection guarantees against even the most motivated adversaries who can prod and poke the actual hardware components of a computing system [23, 31, 43]. Most commonly, software applications are executed on personally owned and operated computing systems. When it comes to issues of IP, licensing, and tamper-resistance, it would be naive for application designers to expect full cooperation from the owners of the systems that run the application. Since these owners have physical access to their system, an application inventor cannot expect any strong security assurances unless the system is based on a physically secure architecture. We chose the AEGIS physically secure architecture [50] as the focus of our work, however other architectures exist that offer similar physical security promises [35].

The AEGIS architecture protects intellectual property privacy and secures software licensing through application code encryption. Every AEGIS processor contains a unique secret which can be used to decrypt encrypted applications on the fly, without revealing the true application code to anyone. For example, each processor may contain a private symmetric encryption key which can be shared with a software designer. The designer can then encrypt his entire application with one processor's key, and make the encrypted application publicly available. The AEGIS architecture guarantees that only that single processor will be able to execute the encrypted application. The architecture also contains integrity verification techniques that provide application tamper-resistance. More details on how AEGIS processors work can be found in [67, 68].

If one is agreeable to a licensing scheme that ties an application to a physical processor, the AEGIS architecture appears to satisfy the three major concerns described. However, the architecture cannot offer these encryption and integrity verification mechanisms without a cost. There is a performance overhead for any application that uses these mechanisms (that is, an application operating in a secure mode). Further, requiring an application to be encrypted *as a whole* can be restrictive for designers, prohibiting the use of shared libraries, amongst other things.

1.2 Partitioning

The AEGIS architecture remedies these failings by allowing small, disconnected portions of an application to execute in a secure mode, while the remaining regions execute in an unencrypted, insecure mode. Allowing an application to switch between these modes during operation leads to an interesting design problem. Tamper-resistance can only be assured for the secure regions of the application. Application code encryption exists for these same small regions, however it is unclear if the same IP privacy assumptions hold. Finally, the licensing scheme used by AEGIS is intended to bind an application to a processor, but a partitioned application really only binds secure regions of an application to a processor. Therefore, a designer's problem is how to partition an application into secure and insecure regions so as to maximize performance while still maintaining tamper-resistance, IP privacy, and license protection. Ultimately, the designer would like to ensure that no one can recreate his application through dissection of his code, that only authorized consumers can use his application, and that it is impossible to modify the way that his application runs for malicious purposes.

1.3 Goals of Work

This work takes a look at what properties are required of a partitioned application to ensure the secrecy of small regions of application code, and to prevent application execution by unauthorized parties. Importantly, these properties focus on the relationship between the partitioned regions of code and the application as a whole, since this is what matters for an actual attack. The specific contents of a partitioned region is independently of little consequence.

Specifically, we analyze one type of attack, given a general but manageable adversarial model, and put forth one practical attack implementation. We then propose metrics that can be used to identify whether a partitioned region of application code can be considered private even in the face of attack. This is not at all clear since, as we show, it is possible to determine the functionality of application code within a small hidden region simply from its interaction with the rest of the application. We also discuss the issue of license protection for partitioned applications and propose a simple bisection rule which ensures that a partitioned application can still be bound to a single processor.

Tamper-resistance guarantees are outside the scope of this work. Most of the physically secure architectures mentioned already focus on this problem extensively. These architectures formulate quite strong security assurances that can be combined with other work related to information flow to ensure tamper-resistance of partitioned applications.

1.4 Organization

This thesis is structured as follows. Chapter 2 begins with a review of previous techniques that were used to ensure application code privacy and to prevent unauthorized execution.

Chapter 3 introduces the adversarial model that is the basis for our investigation, identifying an adversary who is able to observe the input/output relationship of a hidden region of code. Chapter 4 formally discusses the limitations of any such adversary, and Chapter 5 describes the implementation of an attack that is able to create operation equivalent regions of hidden application code. This implementation requires a number of optimizations, but is able to successfully duplicate the functionality of real-world applications.

To defend against such an attack, Chapter 6 presents a number of tests which an application designer can use to identify vulnerabilities in his partitioned application. Further, Chapter 7 describes the important factors that must be taken into account when an application designer partitions his application, and suggests one simple partitioning scheme. We conclude in Chapter 8.

Chapter 2

Prior Work

Intellectual property protection and software licensing has persisted as an open problem in computer science for decades. Application designers have implemented countless mechanisms that attempt to hide the functionality of instruction code and ensure that only authorized users can execute the application in question.

2.1 Software Secrecy

Gosler [9] was one of the first to examine the problem of software secrecy and the possibility of modifying application code to specifically prevent an adversary from determining its contents. In his work he discussed some of the capabilities of adversaries of his era and suggested a set of goals for software protection. Later studies [12, 13, 15, 41] found that simple techniques can be used by an adversary to decipher the contents of hidden applications. To offer such protection of intellectual property privacy, some have proposed methods of "obfuscating" an original application in ways to create an executable binary that is sufficiently incomprehensible, but still functionally equivalent to the original. Collberg et. al. [24, 39] present an in-depth discussion on obfuscation transforms which can be applied to any given block of application code. These transformations are implemented in their code obfuscating compiler, "`Kava`". Obfuscation can increase the difficulty of an adversary to discover the contents of obfuscation code, but also can increase the execute time required to run such regions of code.

Unfortunately, a theoretical treatment of the obfuscation problem by Barak et. al. [37] proved that cryptographically secure obfuscation is impossible for a large family of functions. Recently, Lynn et. al. [56] and Wee [69] have presented positive results which show that some functions can be obfuscated in a cryptographically sound manner. However, the family of functions discussed are too restrictive to be useful for general application protection. Along a similar vain, Sander and Tschudin [29, 28] suggested a protocol which uses homomorphic encryption schemes to allow an executing algorithm to conceal the values being computed while still performing its intended operation. Sadly, this work is again limited to only a small class of functions.

Alternatively, a cryptographically secure means of protecting the intellectual property privacy of a region of application code is to simply encrypt the application instructions themselves. The decryption and execution of the encrypted instructions is then left up to some trusted source. One of the first to suggest this idea was Kent [2] who identified a physically shielded processor as a trust base. Physical security is important since a motivated adversary could easily intercept decrypted machine instructions if a software-only encryption technique is used. Many others [6, 7, 21] have suggested similar solutions. Methods for secure software distribution has also been examined by Herzberg et. al. [11] and Wilhelm [26]. Although these techniques involve encrypted application code and a trusted computing system, they do not satisfactorily detail how to design a physically secure trust base.

More recent architectures, such as XOM [35, 48, 49] and AEGIS [50, 67, 68], remedy this by reducing the size of the trusted base to a single chip and introducing architectural mechanisms that protect the integrity and privacy of off-chip memory. With this, the XOM architecture and AEGIS processor allows applications to be encrypted and executed in a physically secure manner. These architectures follow a similar line of thinking as other recent works that propose specialized hardware modifications for security purposes [25, 30, 52, 70].

Even though these systems encrypt the contents of application code during execution, it is still possible for an adversary to discover the contents of the application code through its use of memory addresses. Address obfuscation [10, 22] is one method which defends against such a discovery. Address obfuscation applies a set of transformations to application code regions that hinder an adversary's abilities to reconstruct the contents of hidden code. An alternative approach is to simply design the application in a way so that no application information is leaked. Agat [33] has suggested a few methods which remove the specific problem of timing leaks. Zhuang et. al. have also suggested additional hardware mechanisms that help protect against leakage [58].

2.2 Software Piracy

There have also been a large number of techniques proposed to handle software licensing. Watermarking, renewability, online-verification, encryption, and authentication have all been suggested to prevent application execution by unauthorized parties. In general, there are only two tactics employed to prevent software piracy, application alteration and execution protection.

Watermarking, renewability, and online-verification all fall under application alteration.

Watermarking [38, 39] is a technique that affixes an indelible signature to an application which uniquely identifies that application, even after an adversary attempts to mutate a duplicated version. This tactic then relies on legal action to thwart an adversary's attempts to use the application in an unauthorized manner. The concept of renewability [40] effectively creates an application expiration date and online-verification [60] requires an application to contact a remote server to run correctly. An idea similar to online-verification is that of a "dongle," [63] which is an individual hardware element that contains some secret and is externally attached to a computing system. During execution of a protected application, special calls are made which require that the dongle be attached to the system.

One of the fundamental problems with application alteration techniques is that they add extra, otherwise useless content to the application code. Consequently, although it may be extremely difficult, it is almost always possible for a motivated adversary to remove this extra code.

Execution protection looks to guarantee that a certain application can only *execute* on a particular computing system. This requires a specialized architecture which is able to execute an encrypted application, or authenticate the integrity of an application using some secret. The XOM architecture [35, 48], the AEGIS processor [68] are both examples of such specialized architectures. In most cases, execution protection is guaranteed by a check made to identify the contents of an application and determine if the application is authorized to run on that particular system and operating system. These architectures also allow application code to be encrypted such that only one particular system can use its secret to decrypt and run the application. TPM from TCG [55], SP-processors [61], and Next Generation Secure Computing Base (NGSCB) from Microsoft [46, 64] are other architectures that add specialized hardware to guarantee the authenticity of applications under execution, however they do not offer the same privacy guarantees as XOM or AEGIS.

2.3 Program Partitioning

The act of partitioning an application for security reasons is not a new approach. Although, to our knowledge, there has not been any work that considered application operation as a whole when analyzing both intellectual property protection and software licensing. Yee [20] discussed the possibility of partitioning applications for copy protection, but did not analyze this problem in depth. White and Comerford [14] developed the ABYSS architecture which allows for secure software partitioning, but only offer a high-level discussion of possible characteristics that can impact the security of a partitioned application. Zhang et. al. [53] suggest program slicing as a means to prevent software piracy but mainly focus on compiler modifications to enable secure partitioning. Their work focuses on efficient transformations that determine how to partition an application which best prevents software piracy. To do this they analyze the contents of candidate partitioned regions and do not consider the possibility that privacy may not be guaranteed. Our work investigates the problem of privacy protection in general, using a model that looks at application protection as whole and treats partitioned regions of code as "black-boxes." A couple sources indicate that the company "Netquartz" [65] also works on similar techniques, but it is difficult to obtain information explaining their exact technology. Most of the remaining work related to application partitioning examines the protection of program user data and the protection of information flow.

"Privtrans" is a tool created by Brumley and Song [54] that automatically separates program source code into two executables, a monitor and a slave. The monitoring code operates within a secure computing environment (a secure execution mode), while the bulk of the application is designated as slave code and is permitted to do whatever it desires. The monitoring code exists to periodically perform checks on the slave code to ensure that it is behaving honorably.

"Secure program partitioning" has been presented by Zdancewic et al. [45] as a language-level technique to protect data privacy. Their work focuses on compiling a single program to run in a distributed computational environment, while still enforcing confidentiality policies on information flow. To this extent, secure program partitioning focuses on guaranteeing end-to-end information flow security.

Finally, a number of architectures *allow* an application to be partitioned [14, 35, 68], but make no guarantees of protecting the privacy of data flowing between partitions. In this regard, information flow techniques offer a more comprehensive solution.

2.4 Application Complexity

The insecurity indicators presented later in this work (Chapter 6) are basically an analysis of the complexity of program code. Over the years many indicators have been proposed that analyze software complexity, however, the majority of this work has involved metrics which are meant to aid in the design process of an application or to improve the quality of software [1, 3, 4, 5, 8, 17, 18]. These software engineering metrics have little security value.

Yang et. al. [27] is one of the only investigators that constructs software complexity metrics intended to identify how easy it is to de-construct and comprehend applications. Unfortunately, their work does not focus on the security implications of these metrics, nor does it handle malicious adversaries.

Chapter 3

Modeling an Attack

This work seeks guarantees that affirm the secrecy of the functionality of private regions of partitioned applications and prohibit unauthorized use of these applications as a whole. To this end, our principal concern is an adversary who wishes to reconstruct a "*counterfeit*" region of code that is able to emulate the functionality of an "*authentic*" region of private code within a partitioned application. This adversary could use his counterfeit reconstruction to perform any of the same tasks that the authentic region performed in the original partitioned application. Further, the licensing scheme we have proposed is contingent upon absolute confidentiality of the partitioned "private" regions of code. If an adversary knew the contents of the private regions, he could simply replace these regions with his own public code.

3.1 Application Operation Equivalence

One crucial observation in this work is that most adversaries are only interested in running a *single* partitioned application APP under a limited set of usage scenarios. These usage scenarios can be defined by the set of an application's external inputs over time, $\langle \mathbf{\Lambda} \rangle$, called the application's "workload."

To achieve this goal, an adversary does not need to reconstruct an *exact* copy of all authentic private regions within an "authentic application." All he must do is construct counterfeit regions of code that replace authentic regions of code to create a "counterfeit application" that operates the same as the authentic application. For example, imagine a region of private code in an application that performs the "power" function $f(x, p) = x^p$. If the public regions of that application only ever need to determine the cube of x, then a reconstructed counterfeit region of code would similarly only ever need to compute the cubic function $f(x, 3) = x^3$. Since the adversary is only interested in running that particular application on a specific workload, there is no benefit in replacing the private region of code with a true power function. All that matters is the ultimate functionality of the entire application. We refer to this concept as *Application Operation Equivalence* (AOE).

Definition 1.

(Basic) Application Operation Equivalence: AOE($APP^{Auth}, APP^{Cf}, \mathbf{\Lambda}$)

Given an authentic version of an application APP (APP^{Auth}), and a counterfeit version of APP (APP^{Cf}), APP^{Auth} and APP^{Cf} are AOE if the set of outputs of both applications, $\mathbf{\Psi}^{Auth}$ and $\mathbf{\Psi}^{Cf}$, are exactly equivalent when run on one set of inputs to the application, $\mathbf{\Lambda}$.

Again, this definition differs from *functional equivalence* since the outputs of an application must only match for a single set of inputs $\mathbf{\Lambda}$, not all possible inputs. Further, the inputs and outputs of a private region of code ($\boldsymbol{\lambda}$ and $\boldsymbol{\psi}$ respectively) are only of indirect consequence since $\boldsymbol{\psi}$ may or may not effect $\mathbf{\Psi}$. Figure 3-1 shows this relationship between a private procedure and an entire application.

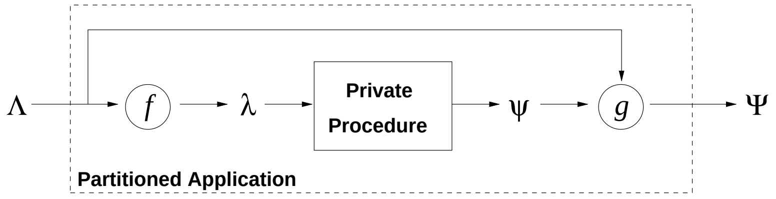


Figure 3-1: Partitioned application inputs and outputs.

Ideally, an adversary would prefer his counterfeit application to be AOE for absolutely every possible input. However, it is usually impossible to know every conceivable input to an application. Practically, what an adversary wants is for his counterfeit application to operate equivalently for *as long as possible*. Therefore, we introduce the concept of *Temporal Application Operation Equivalence* which includes time in its definition.

Definition 2.

Temporal Application Operation Equivalence: T-AOE($APP^{Auth}, APP^{Cf}, \langle\mathbf{\Lambda}\rangle, t_s, \omega$)

Let us assume that two applications APP^{Auth} and APP^{Cf} begin execution at time 0 and finish execution at the same time H. At each step in time t both applications are given one set of inputs $\mathbf{\Lambda}_t$ taken from a set of many sets of inputs, the workload $\langle\mathbf{\Lambda}\rangle$. These applications are Temporally AOE for the time ω if, for the time period $[t_s, t_s+\omega]$, the outputs of both applications $\mathbf{\Psi}_t^{Auth}$ and $\mathbf{\Psi}_t^{Cf}$, are exactly equivalent (assuming $(t_s + \omega) \leq H$).

Given this definition, we say that an adversary's goal is to create a counterfeit private region of code for a specific partitioned application that maximizes the T-AOE time ω (ideally, $\omega \to \infty$). This matches the goal of an adversary examining an authentic partitioned application and recreating his own counterfeit application which serves the same "utility."

3.2 System Model

Here we describe the computing systems that this work assumes all partitioned applications will be run on. We first propose a system that consists of one insecure processor and one secure coprocessor working in tandem to execute an application. We believe that this secure coprocessor model is one of the best designs for ensuring procedure privacy and software license protection. Unfortunately, no secure coprocessor architectures [19, 20, 32] have been implemented that adhere to our description. Therefore we present a specific usage strategy of the AEGIS secure processor [68] that is *equivalent* to our desired secure coprocessor model. By using an existing, fully-implemented architecture in a manner that agrees with our secure coprocessor model, we can perform realistic experiments and make insights that have a meaningful and immediate impact on real systems. Unless otherwise stated, future chapters in this work assume an AEGIS secure architecture model.

3.2.1 Secure Coprocessors

The secure coprocessor model assumes a computing workstation with one or many fast, standard processors, and a single, relatively slower secure coprocessor which may or may not be removable. The coprocessor is an implementation of a physically secure architecture and contains a cryptographic secret. It also contains a small scratch-pad memory that , like the rest of the coprocessor, is impervious to physical monitoring or attack.

A removable coprocessor would allow a single secure chip to be used on multiple different workstations, although not at the same time. For example, a "dongle" containing the secure coprocessor can be carried by a human user as he moves between workstations. In this way the cryptographic secret within the coprocessor is tied to a human user and not a specific computing system. While this binding can have many benefits, it is unclear whether it outweighs the added hassle of carrying a dongle.

Figure 3-2 describes our secure coprocessor setup. Briefly stated, all computation done within the secure coprocessor, as well as all data within a secure coprocessor's internal memory cannot be monitored by an adversary. All processors do however share a global memory that is completely observable and controllable by an adversary. A partitioned application keeps all of its application code in global memory, but encrypts private portions of the code using a key which is specific to a single secure coprocessor.

An application starts execution on the standard processors, where public, unprotected code can be run. From time to time, the coprocessor is issued a directive from the standard processors to begin executing encrypted application code at a specific address of shared (untrusted) memory. The encrypted application code can be thought of as a "procedure" that is called by unencrypted code running on the standard processors. The inputs to an encrypted procedure are values in shared memory that are read during private execution.

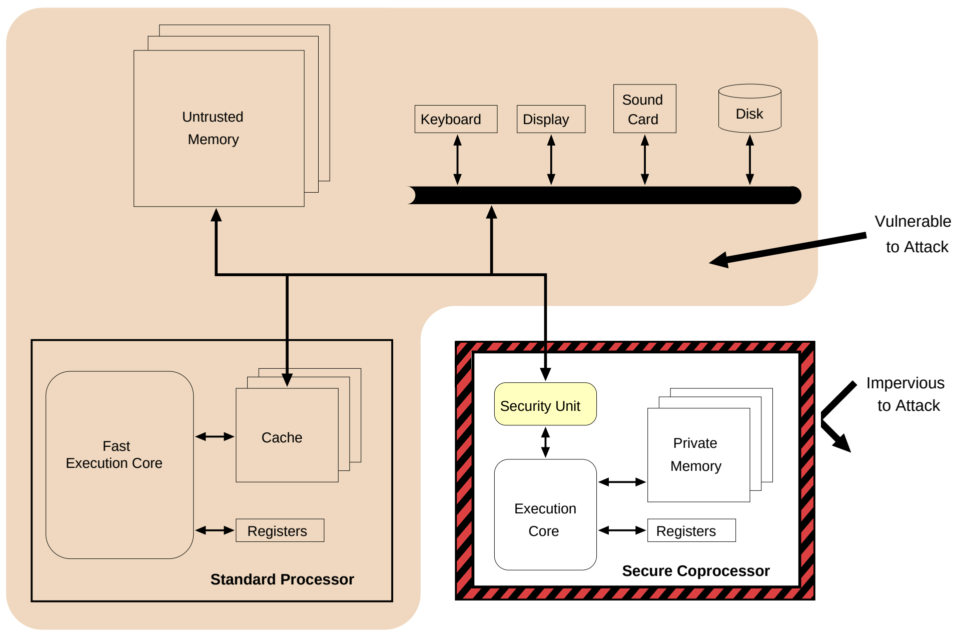


Figure 3-2: Secure coprocessor model.

Similarly, the outputs to an encrypted procedure are values that are written into shared memory during private execution.

The designated encrypted procedure is read by the coprocessor, authenticated, decrypted, and then executed within the coprocessor where it uses its own scratch-pad memory to hold a private stack for secure execution. The private procedure can perform reads and writes to shared memory as necessary, however these accesses are *not* private. (Therefore the inputs and outputs of a procedure can be monitored.) Once the private procedure reaches termination, it returns a signal to the standard processors that it has finished. Alternatively, the encrypted procedure may need to call other public or private procedures. In this case the encrypted procedure suspends its operation, saves what it must onto the coprocessor's internal stack, and makes a request to execute a different procedure with an explicit flag signalling that control should return to the coprocessor after that procedure has finished executing. Similarly, recursion is possible through use of the coprocessor's internal stack.

To simplify our model, we assume that multiple private procedures cannot share any state information. When a private procedure executing on a secure coprocessor calls another public procedure, it is clear that data can only flow from the private procedure to the public procedure through untrusted shared memory. However, when a private procedure calls another private procedure (or itself), it may be possible to share data between the two procedure calls using the internal coprocessor stack, bypassing the untrusted global memory. We do not allow this because, abstractly, one can consider these two procedures to really be one procedure with a more complicated structure. Therefore, this work will assume that an adversary can observe all data flows *between* separate private regions of code. We also assume that the coprocessor does not contain an instruction or data cache that may mask reads and writes. At the termination of any private procedure, a coprocessor cache would need to be flushed to shared memory in any case. The standard processors can have a cache as long as a coherence protocol is implemented between all processors and main memory. Finally, we assume that all applications operate in a single-threaded fashion.

3.2.2 AEGIS

Simply put, the AEGIS secure processor assumes that all data within the processor chip is trusted, and all data that leaves the processor chip is untrusted (namely, off-chip memory). As shown in Figure 3-3, the processor itself is encased in a physically secure packaging which conceals all processor information such as the currently executing instructions, data and instruction caches, as well as other architectural units. We assume that it is impossible for an adversary to determine these internals. An adversary cannot ever determine such internals. However, an adversary is able to monitor the contents of off-chip memory, disks, and communications buses.

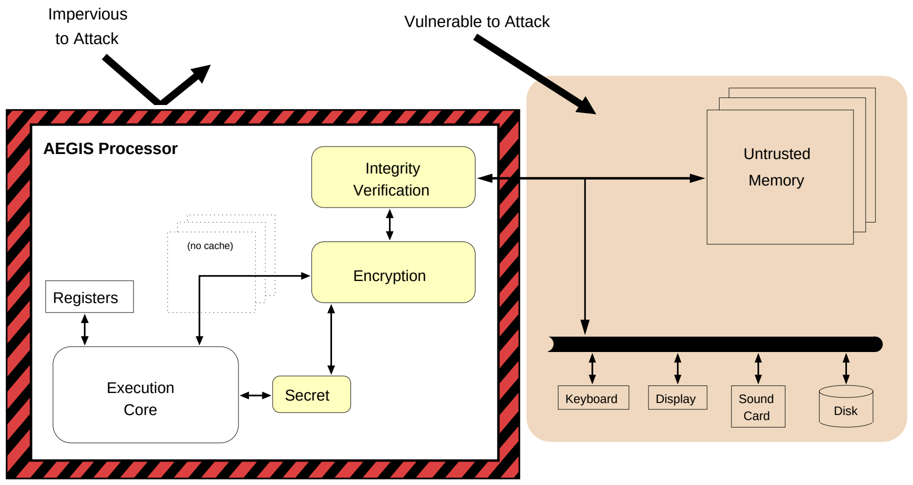


Figure 3-3: AEGIS processor model.

To protect running applications AEGIS provides four separate execution modes offering varying levels of security. These are Standard (STD) mode, Tamper-Evident (TE) mode, Private Tamper-Resistant (PTR) mode, and Suspended Secure Processing (SSP) mode. An application running under STD mode is offered no extra-ordinary security guarantees. When an application runs under TE mode integrity verification mechanisms are turned on that raise exceptions when the application is modified by a third party (either another program or an active physical adversary). An applications running in PTR mode is provided integrity verification as well as data and application code encryption. Finally, applications running in TE or PTR mode are able to transition into SSP mode which offers no security protection mechanisms, but still protects portions of memory previously reserved for other application code regions that execute under TE or PTR modes. At boot-up, the AEGIS processor begins execution in STD mode, however an application can immediately transition into TE or PTR modes when it begins.

The application partitioning methodology presented in this work assumes that an application begins by transitioning into PTR mode. This initial transition into PTR mode only sets up the security keys and immediately transitions to the proper beginning of an application under SSP mode. After that any number of transitions can be made from SSP mode to PTR mode and back when "private procedures" are called. A partitioned application is therefore divided into private procedures which execute in PTR mode and public procedures which execute in SSP mode. Figure 3-4 briefly shows how this partitioning is divided in memory on the AEGIS architecture, although a more in depth description can be found in the work by Suh et. al. [66, 67, 68]. To match the secure coprocessor model, regions of code executing under PTR mode can access an encrypted region of main memory that holds a private stack. Similarly, the inputs to a private procedure are defined by the procedure's reads of shared memory, *as well as* the contents of any system registers that remain constant during a transition from SSP mode to PTR mode. The outputs of a private procedure are defined by the writes to shared memory as well as the values of any system registers that are not zeroed out during the transition from PTR mode to SSP mode.

Again, to simplify matters, we will assume that the state associated with each unique procedure that runs in PTR mode is independent of all others. Figure 3-4 shows this assumption, that, memory cannot be read by one PTR procedure that was written by

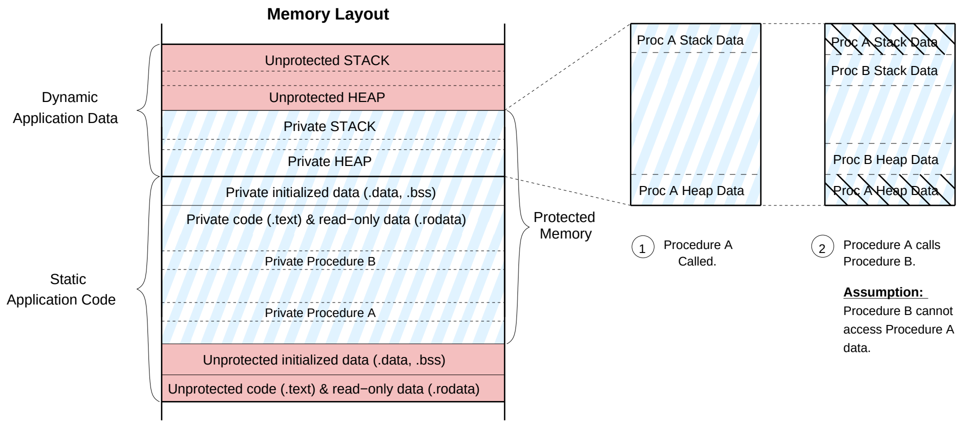


Figure 3-4: AEGIS memory layout of application.

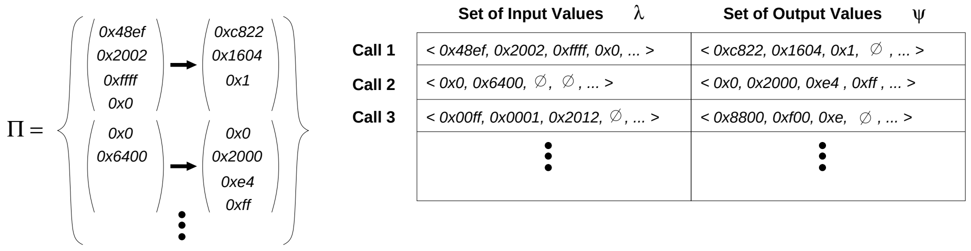
another. We similarly assume that the AEGIS processor does not contain a cache. This has little effect because a cache would need to be flushed on every transition between SSP mode and PTR mode anyways. Finally, we assume that only one thread of execution is possible at any given time. Therefore, once a private procedure has begun execution on the AEGIS processor, no other thread can interrupt it until it has completed operation of its own volition.

3.3 Adversarial Knowledge

Modeling the capabilities of an adversary bent on creating a counterfeit application is a tricky business. This adversary is almost certainly a human agent who is motivated to reconstruct private code through whatever means possible. He can use purely social, "real-world" tactics to discover the contents of the private region in a non-computational fashion. For example, the adversary can talk with the original author, he could *be* the original author, or he can read confidential documentation, and so on. All of this information can be considered innate "knowledge" that the adversary possesses before attempting to reconstruct private regions of code.

3.3.1 Input/Output Relations

Given the vagueness and infinite dimension of such human knowledge, it is unclear whether any concrete model can sufficiently explain a real-world adversary. For this reason, our work treats all private procedures as an abstract function with inputs and outputs that the adversary is capable of observing. The adversaries we deal with are only cognizant of a set of input/output relationship pairs for a given private procedure, and *nothing else*. For example, we do not allow an adversary to use the assumption that a particular private procedure's output is related to its inputs as a function of some combination of "features" (which is a common assumption used in computational "learning" algorithms). The set of input/output relationship pairs is referred to as $\mathbf{\Pi}$ and is depicted in Figure 3-5. It can be thought of as simply an enormous lookup table with a column for inputs and a column for resulting outputs.



	Set of Input Values λ	Set of Output Values ψ
Call 1	< 0x48ef, 0x2002, 0xffff, 0x0, ... >	< 0xc822, 0x1604, 0x1, ∅ , ... >
Call 2	< 0x0, 0x6400, ∅, ∅ , ... >	< 0x0, 0x2000, 0xe4 , 0xff , ... >
Call 3	< 0x00ff, 0x0001, 0x2012, ∅ , ... >	< 0x8800, 0xf00, 0xe, ∅ , ... >
	⋮	⋮

Figure 3-5: Set of input/output relationship pairs $\mathbf{\Pi}$.

We concede that input/output relationship pairs are not the only information an adversary might know about a private procedure in a real-world attack. This set **Π** focuses on the interactions between public and private procedures and ignores interactions between public procedures and each other, which often can reveal the purpose of a private procedure.

For example, let us assume a public procedure is known to handle the "`SpellCheck`" button of a GUI office application, and that public procedure exclusively calls some other private procedure. A panoptic adversary might be inclined to believe that the private procedure in fact manipulated text in some way or performed database lookups for spell-checking. This inclination might aid even further analysis. However, it is unclear whether such a panoptic adversarial model can be constructed without including some aspect of human intuition or prior knowledge which our model explicitly prohibits. Therefore, we find it fair to say that it is sufficient to exclusively focus on the interactions between public and private partitions when working with an adversarial model that excludes such a priori knowledge.

3.3.2 Obtaining Input/Output Pairs

Restricting an adversary to only know the inputs and outputs of a private procedure agrees perfectly with the secure coprocessor and AEGIS platforms described in Section 3.2. These architectures are explicitly designed to hide all computation that occurs within a private procedure. Therefore, the only information visible is the data values that are passed into and

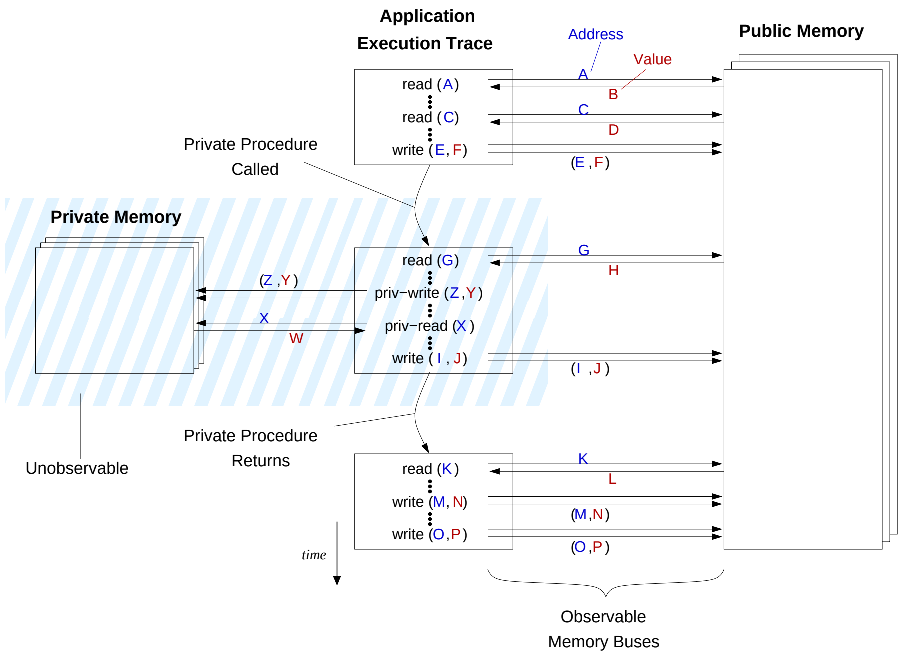


Figure 3-6: Observing inputs & outputs with a secure coprocessor.

out of private procedures. Under both architectures discussed, these data values necessarily reside in public, observable memory before and after a private procedure is executed.

Figure 3-6 details how private procedure inputs and outputs are observable in a secure coprocessor architecture. All partitioned applications begin by executing public regions of application code. Public code is completely observable as well as any of the reads and writes that are made to main memory. Reads and writes are identified by the memory address that is to be read or written to, and the corresponding data value. This identification is called an Address/Value (AV) pair.

When a public procedure calls a private procedure, control flow is transferred to the private procedure running on the secure coprocessor while the public procedure stalls waiting for a return signal. Execution within the secure coprocessor cannot be seen, including any use of private memory. However, an adversary can observe all reads and writes to public memory by monitoring the unprotected main memory address and data bus. After a private procedure returns, the set of all values that were read during private execution is called the *input set* $\boldsymbol{\lambda}$, which is indexed by the address of each input value. The set of all written values is similarly called the *output set* $\boldsymbol{\psi}$ with the same indexing. The output set should exclude redundant addresses, so in the case of two writes to the same address, only the chronologically second write is included in $\boldsymbol{\psi}$. These two sets $\boldsymbol{\lambda}$ and $\boldsymbol{\psi}$ form one input/output relationship pair $(\boldsymbol{\lambda}, \boldsymbol{\psi})$ for the private procedure. An adversary can construct

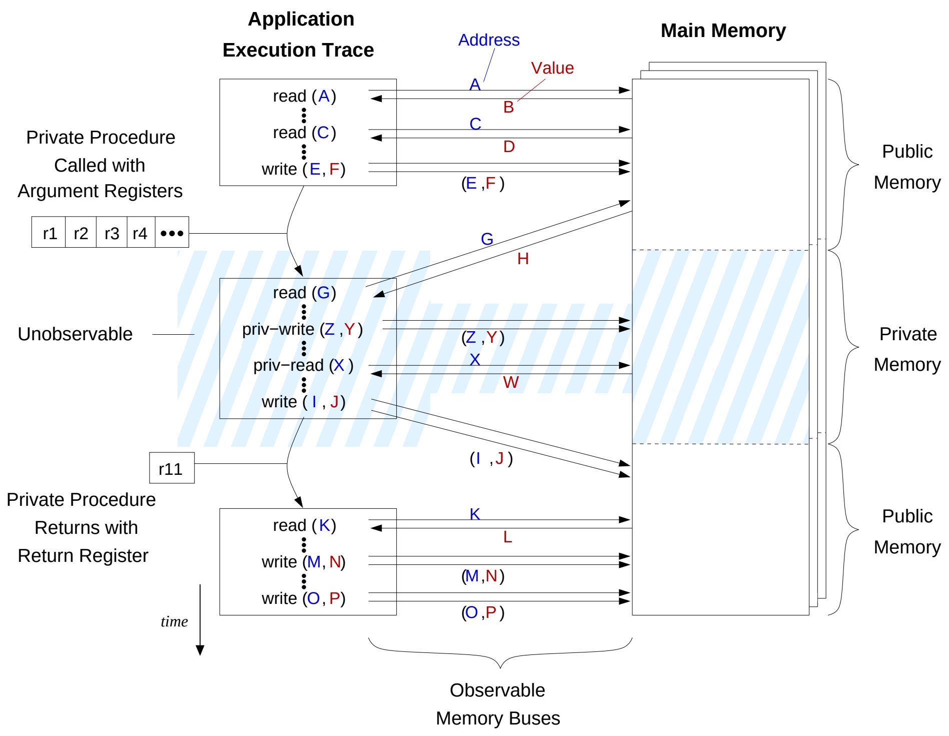


Figure 3-7: Observing inputs & outputs with AEGIS.

a set of input/output relationship pairs $\mathbf{\Pi}$ by observing many calls to the private procedure.

Figure 3-7 details how private procedure inputs and outputs can be observed in the AEGIS architecture. As with a secure coprocessor, public code is completely observable including all reads and writes to main memory. Similarly, an adversary can monitor the AV pairs of reads and writes a private procedure makes to public memory. It is worth note that in the AEGIS architecture, reads and writes to private memory appear on the (public) main memory buses. Although the data is securely encrypted [51], the addresses are not, and therefore information may be leaked about the private procedure. Since it is possible to avoid such information leakage [10, 22, 33], for simplicity this work assumes that reads and writes to private memory reveal zero information, as in the secure coprocessor case.

Unlike the secure coprocessor model, when a private procedure is called, input arguments can also be passed to the procedure via argument registers. These argument register values must be added to the inputs set $\boldsymbol{\lambda}$ along with any read values. In this case the register name can be used to index into $\boldsymbol{\lambda}$. Similarly, an output register can be used by a private procedure to return an output value. This output register value must also be added to the outputs set $\boldsymbol{\psi}$.

3.4 Monitor & Swap Attack

The actual attack this work will focus on is called the "*Monitor & Swap Attack*." This attack describes the specific steps an adversary walks through to determine the input/output relationship pairs of a function and to construct a counterfeit application. Although there are countless ways in which an adversary can attack an application, we chose this as one of the simplest and possibly the most common means of attack. The basic Monitor & Swap Attack is portrayed in Figure 3-8 and can be applied to either a *passive* or *active* adversary.

As it sounds, a passive adversary performing a Monitor & Swap Attack merely observes an authentic partitioned application running on one of the secure architectures described. During this time the execution of unencrypted public procedures is completely observable since these procedures execute in an insecure mode. The public application code can make calls at any time to execute encrypted private procedures. Before and after any call is made to execute a private procedure, the adversary can see the entire contents of memory and the state of system registers. Again, during the execution of the private code, only reads and writes to public memory are visible. The state of memory is read before and after the procedure call to determine the values within $\boldsymbol{\lambda}$ and $\boldsymbol{\psi}$, while the monitored reads and writes identify the addresses used to index the sets.

An active adversary does not only observe an existing application, but can execute a private procedure himself, using whatever inputs he desires. To do this he can write wrapper code that fills a region of memory with data and then calls the private procedure. The input

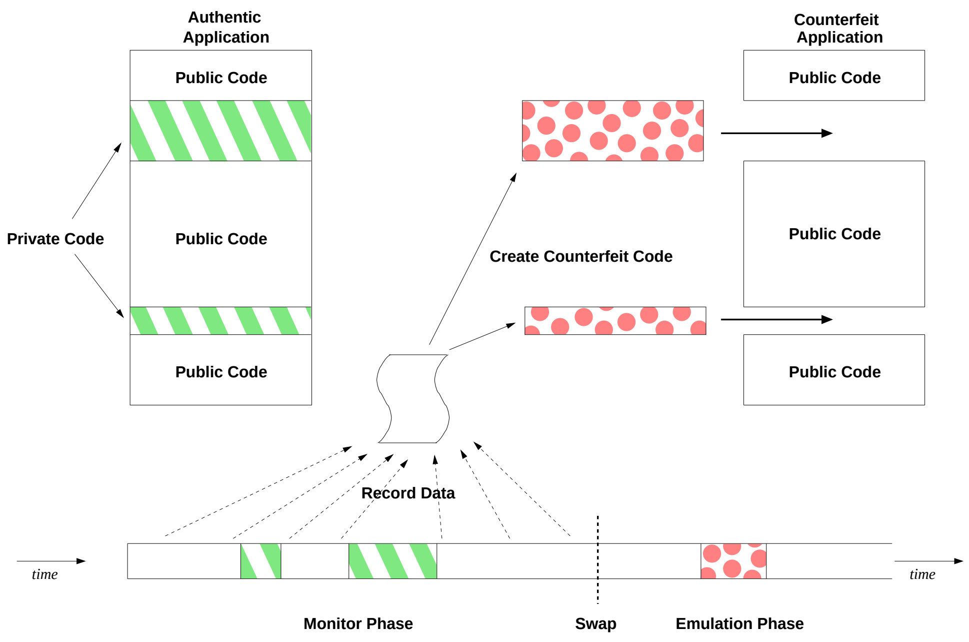


Figure 3-8: Monitor & Swap Attack.

and output sets $\boldsymbol{\lambda}$ and $\boldsymbol{\psi}$ are filled in the same way as in the case of a passive adversary, except the active adversary clearly has more control over the contents of $\boldsymbol{\lambda}$.

After some time of monitoring private procedures, be it actively or passively, the adversary performing this attack "swaps" the authentic private procedures with his own counterfeit private procedures and continues to use the application normally. During this "emulation phase," the adversary's counterfeit procedures simply emulate the functionality of the authentic procedures during any private procedure call. The adversary's goal is to create a set of counterfeit procedures such that the new counterfeit application is T-AOE with the authentic application for the greatest time ω (T-AOE$(APP^{Auth}, APP^{Cf}, \boldsymbol{\Lambda}, t_s, \omega)$). Here the input set $\boldsymbol{\Lambda}$ is any application input which is applied after the swap time t_s. Under a Monitor & Swap Attack, this time ω can be considered the "time-till-failure" since it represents how long an application can continue running normally using counterfeit procedures.

This swapping scenario corresponds with the possible removal of a "dongle" type secure coprocessor from a system, or it can correspond to a secure (co)processor changing its secret. Generally speaking, this attack is also analogous with an adversary creating a new counterfeit application and executing it on a different system. Whether the adversary continues or restarts an application from the beginning is irrelevant, only the time of correct operation ω and the input set $\boldsymbol{\Lambda}$ matters. Therefore, we will assume that the adversary swaps private procedures and continue to use the application as he has since this appears to be the slightly harder case.

Chapter 4

Adversarial Capabilities

Given the model of attack discussed, it would be useful to know the limitations of an adversary who attempts to succeed in creating and executing a counterfeit application. Principally we would like to know the probability of success for an adversary with unlimited resources since this describes an adversary's optimal abilities.

4.1 Definitions

To begin, let us imagine an application APP with a single private procedure PP that takes a maximum of q inputs and returns a maximum of s outputs. Each of these inputs and outputs are values taken from some discrete space $\mathbb{V}$ where $|\mathbb{V}| = \kappa$ and $\mathbb{V} \subset \mathbb{Q}$. In a real-world computing system, $\mathbb{V}$ often represents binary values where $\kappa = 2^{32}$ or $\kappa = 2^{64}$, and so on. An indexing address is also associated with every input and output and can be any cardinal number between 0 and 2^a.

Inputs and outputs of the application as a whole, $\mathbf{\Lambda}$ and $\mathbf{\Psi}$, are simply abstract values. Each of these values is chosen from the set of all possible application inputs, $\widehat{\mathbf{\Lambda}}$, and the set of all possible application outputs, $\widehat{\mathbf{\Psi}}$. This abstraction agrees with real-world applications that may input any value from alphanumeric text to mouse-clicks to the temperature, and which may output anything from text to images to sound.

As can be seen in Figure 4-1, the vector of inputs to the private procedure $\boldsymbol{\lambda}$ are deter-

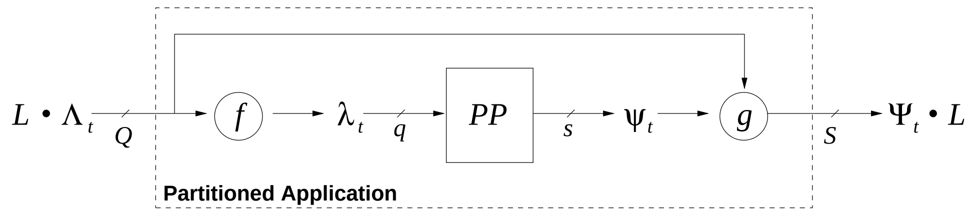


Figure 4-1: Partitioned application attack model.

mined by the vector of inputs to the application $\mathbf{\Lambda}$ applied to the function f. Similarly, the vector of outputs of the private procedure $\boldsymbol{\psi}$ helps determine the outputs of the application $\mathbf{\Psi}$ through the function g. (The function g subsumes any public procedures that may also use the application inputs $\mathbf{\Lambda}$.)

Although $\boldsymbol{\lambda}$ includes an AV pair row for every address that *can* be read by PP, each individual call to a private procedure may only read a few of these addresses as an input. Therefore, we define $R(\boldsymbol{\lambda})$ to be the rank of the vector of inputs $\boldsymbol{\lambda}$, or the number of AV pairs that are actually "used" by a single call to the private procedure. The unused elements of $\boldsymbol{\lambda}$ (addresses that are not read) are simply filled with $\varnothing$. $R(\boldsymbol{\psi})$ is defined in the same way.

Now let us imagine a passive adversary A_p who wishes to create a counterfeit version of the application APP, APP^{Cf}, from the authentic version APP^{Auth}. A_p begins monitoring an execution of APP^{Auth} at time $t = 0$. Between time $t = 0$ and time $t = t_s$, L different sets of inputs are applied to the application from $\langle \mathbf{\Lambda} \rangle$.

According to our model, the adversary A_p is able to observe the input/output relationship pairs $\mathbf{\Pi}$ during this time. By time t_s, the table $\mathbf{\Pi}$ is made up of L rows, each containing a pair $(\boldsymbol{\lambda}, \boldsymbol{\psi})$. If we assume that each value within $\boldsymbol{\lambda}$ and $\boldsymbol{\psi}$ is identified by $\lg(\kappa)$ bits, then the size of one row entry in $\mathbf{\Pi}$ is

$$\textit{Row Size } \theta \quad = \quad (\lg(\kappa)(q + s)).$$

This tells us that the size of $\mathbf{\Pi}$ at time t_s is

$$\textit{Size}(\mathbf{\Pi}) \quad = \quad L\theta.$$

On way to describe describe the absolute maximum size of $\mathbf{\Pi}$ is in terms of the set of all possible application inputs and application outputs, $\widehat{\mathbf{\Lambda}}$ and $\widehat{\mathbf{\Psi}}$. However, according to our assumptions, the function f can only produce a maximum of $(\kappa \cdot q)$ unique outputs from even an infinite set of inputs $\widehat{\mathbf{\Lambda}}$. Therefore $L \leq \kappa q$ and the maximum size of $\mathbf{\Pi}$ is

$$\textit{MaxSize}(\mathbf{\Pi}) \quad = \quad \kappa q \theta.$$

An active adversary A_a requires no changes to these definitions. In the case of an active adversary, a set of inputs for a private $\boldsymbol{\lambda}$ are chosen during the time 0 and t_s. Therefore an application input set that occurs at time t, $\mathbf{\Lambda}_t$, can only be found by inverting the function f, $f^{-1}(\boldsymbol{\lambda}_t) = \mathbf{\Lambda}_t$. We still say that L different sets of inputs have been applied from $\langle \mathbf{\Lambda} \rangle$.

4.2 Success Probability

To determine the probability of success for an adversary, we must derive a set of experiments that describe the stochastic process that an adversary undergoes when performing an attack. As previously stated, an adversary is said to succeed when he is able to create a counterfeit application that is T-AOE with its corresponding authentic application for at least some time ω.

Let us examine the chances of an application APP^{Auth} being AOE to APP^{Cf} on some set of application inputs $\mathbf{\Lambda}_t$ seen after time t_s in a Monitor & Swap Attack. This corresponds to a single "call" of the application during the emulation phase of the attack and requires an adversary to emulate one call of a private procedure given the inputs $\boldsymbol{\lambda}_t$. We ignore the possibility of a single set of inputs $\mathbf{\Lambda}_t$ causing a private procedure to be invoked multiple times since this can be abstractly thought of as one call.

Such a call can be seen as a Bernoulli trial where it is either true or false that the two applications are AOE at time t (keeping with our assumption that the procedure does not retain state between calls). If we can determine the probability of success for this single trial, P_{call}, then the probability of creating a counterfeit application that is T-AOE for at least some time ω is simply a sequence of successful Bernoulli trails. The overall success probability P_ω is therefore

$$P_\omega \quad = \quad (P_{call})^\omega .$$

4.2.1 Trial Success P_{call}

The probability of successfully emulating a procedure once depends on the input/output relationship pairs set $\mathbf{\Pi}$. This is because we explicitly ignore any further knowledge which most often can only be supplied by human intuition.

The best possible adversary is one who has infinite memory, computational power, and time. With unlimited computation time, he can feed every possible input to any private procedure (κq unique inputs) and record every output of the procedure in his table $\mathbf{\Pi}$. This creates a $\mathbf{\Pi}$ table of maximum size ($\kappa q \theta$). Since the private procedure is a deterministic function, and the adversary has *seen* the corresponding output to every possible unique input, his probability of success is $P_{call} = 1$.

A slightly restricted adversary only has infinite memory and computational power. He must, however, limit the amount of time he can spend monitoring an authentic application. Since the computation that is performed within the private procedures of an authentic application is done by a secure processor that does not have infinite computational power, this limits the number of inputs an adversary can observe being fed to a private procedure.

Let us assume that this adversary is able to monitor an authentic application for a very long time t_s, observing a large number of sets of application inputs L, and remembering *all* input/output relationship pairs $(\boldsymbol{\lambda}, \boldsymbol{\psi})$ in $\mathbf{\Pi}$. One call of the application during the emulation phase, given an input $\mathbf{\Lambda}_t$, will result in one of two cases.

In the first case, the results of $f(\mathbf{\Lambda}_t)$ is an input vector $\boldsymbol{\lambda}_t$ that already exists in some row of the table $\mathbf{\Pi}$. This means that the adversary has already seen this input/output relationship pair. To emulate the private procedure, he simply finds the row with a first column matching $\boldsymbol{\lambda}_t$ and reproduces the outputs $\boldsymbol{\psi_t}$ (and $\boldsymbol{\Psi_t}$). These must be equivalent to the outputs generated by an authentic application since they are simply a copy of what was already observed in the deterministic private procedure. Consequently, under the first case, $P_{call}^{(1)} = 1$.

In the second case, the result of $f(\mathbf{\Lambda}_t)$ is an input vector $\boldsymbol{\lambda}_t$ that does *not* exist in any row of the table $\mathbf{\Pi}$. This means the adversary has not seen these inputs before. Given our assumptions that the adversary knows nothing except input/output relationship pairs, there is little else to do but to "guess" the values of the outputs set $\boldsymbol{\psi}$. Lacking any knowledge or assumptions of the true distribution of output values, he can only speculate about what the underlying output distribution is given all of the sets of outputs seen in the second (output) column of $\mathbf{\Pi}$. If the true distribution of output values is uniform, then the adversary's chance of success is $P_{call}^{(2)} = \left(\frac{1}{\kappa}\right)^s$.

If we assume the inputs sets $\mathbf{\Lambda}_t$ are selected from $\widehat{\mathbf{\Lambda}}$ uniformly, then the probability of success P_{call} is

$$P_{call} = \left(\frac{L}{|\widehat{\mathbf{\Lambda}}|}\right) + \left(1 - \frac{L}{|\widehat{\mathbf{\Lambda}}|}\right)\left(\frac{1}{\kappa}\right)^s$$

4.2.2 Memoization

The above analysis shows us that even an adversary with unlimited memory and computational power can still do no better than "memoizing" every input/output relationship pair he observes. Therefore, the remainder of this work will focus on adversaries who perform "Memoization Monitor & Swap Attacks." As it sounds, this type of attack consists of an adversary recording all input/output relationship pairs during the monitoring phase, and then using those input/output relationship pairs to create a counterfeit application to use during the emulation phase. By investigating Memoization Attacks, we can cover all possible security threats given the assumptions we have made in on our model. Algorithm 1 gives basic pseudo-code describing how any Memoization Attack would emulate a given private procedure PP. The variables and cases listed correspond with those used in Sections 4.1 and 4.2.1.

Algorithm 1 (Procedure emulation under a Memoization Attack).

```
$Emul_{PP}(\boldsymbol{\Pi}, \boldsymbol{\lambda})$ : Case (1): $\exists\ (\boldsymbol{\lambda}', \boldsymbol{\psi}') \in \boldsymbol{\Pi}$ s.t. $\boldsymbol{\lambda} = \boldsymbol{\lambda}'$
                                  return $\boldsymbol{\psi}'$
                      Case (2): $\forall\ (\boldsymbol{\lambda}', \boldsymbol{\psi}') \in \boldsymbol{\Pi}$, $\boldsymbol{\lambda} \neq \boldsymbol{\lambda}'$
                                  Let $\boldsymbol{\psi}^R$ = new vector sized $s$
                                  $\forall\ j \quad \boldsymbol{\psi}^R_j \overset{\mathfrak{R}}{\leftarrow} (\mathbb{V} \cup \varnothing)$
                                  return $\boldsymbol{\psi}^R$
```

Chapter 5

Implementing a Memoization Attack

It is indeed possible for a realistic adversary to use a Memoization Monitor & Swap Attack to successfully create a useful counterfeit application. To illustrate this point, we implemented a real-world attack on the AEGIS architecture which was realistically constrained by the size of the input/output relationship pairs table ($\mathbf{\Pi}$) and by the amount of time allowed for the monitoring phase. Although at first glance such an implementation may seem simple, we discuss here some of the problems and solutions of what is actually a complicated attack. To confirm the functionality of our Memoization Attack implementation, we treated real programs from the SPEC CPU2000 [34] benchmark suite as if they were partitioned applications. The results show that it is possible to create a counterfeit procedure that is Application Operation Equivalent to the original using only the procedure's interaction table. (It is worth noting that any such attack is still highly dependent on the application partitioning scheme used, as will be discussed in Chapter 7.)

5.1 Handling Input Self-Determination

One of the chief problems involved in constructing a Memoization Attack is determining the inputs of a given private procedure. Chapter 3 defined the set of inputs to a private procedure as a vector $\boldsymbol{\lambda}$ containing data values indexed by addresses. The trouble is, for any specific call to a private procedure, many of the indexes of $\boldsymbol{\lambda}$ can contain the value "$\varnothing$" since multiple calls do not always read the same memory addresses. This is because the private procedure itself determines what memory addresses are to be read as it executes. All of the input addresses read (except the first0 can depend upon the values of previously read addresses.

During the monitor phase of an attack this property of self-determination does not complicate an implementation of a Memoization Attack. Every input Address/Value pair that is read from public memory can be observed and appropriately inserted into the input vector $\boldsymbol{\lambda}$. However, when a private procedure is called during emulation the adversary cannot know the complete set of inputs for this particular call. Therefore he cannot find a match within the input/output relationship pairs table $\mathbf{\Pi}$. As shown in Figure 5-1, an adversary can only know the first input address and value which the procedure takes at the time of a call. He cannot be certain of the next address that should be fed as an input since that is determined by some hidden mechanism within the private procedure.

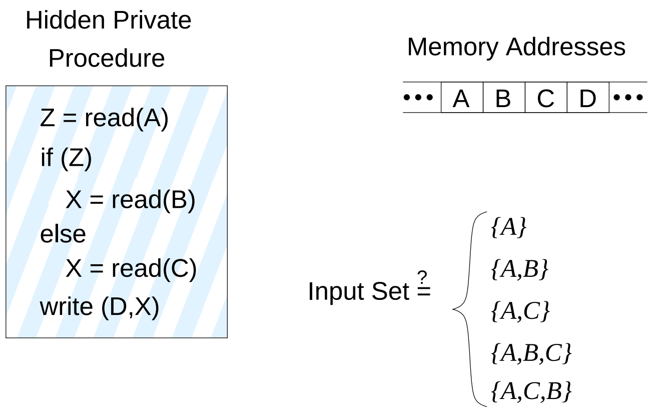


Figure 5-1: Uncertainty of inputs on private procedure call.

Naively, this problem can be answered by placing the entire contents of public memory into the vector $\boldsymbol{\lambda}$ on every private procedure call. With this, the first column of each row in $\mathbf{\Pi}$ can be matched against the current contents of memory whenever a private procedure is called during emulation. This will certainly work, however the $\boldsymbol{\lambda}$ vectors contain an enormous amount of unimportant data. Due to memory space limitations, any realistic adversary must find a way to determine private procedure inputs in a more efficient manner. Ideally, an adversary would like to only index the vector $\boldsymbol{\lambda}$ using the set of addresses that a private procedure *can* read, and would like to set all addresses which are not used for a given procedure call to $\varnothing$.

To solve this problem, this implementation chooses to add another dimension to the table which is constructed during the monitor phase, *temporal ordering*. Instead of an input/output relationships table $\mathbf{\Pi}$, this implementation records an “Interaction Table” $\mathbf{\Xi}$ which keeps track of the ordering and value of inputs and outputs. This interaction table is then used during the emulation phase to identify the correct input/output relationship pairs required.

As shown in Figure 5-2, the table contains one column for each time a specific private procedure is called. The column itself contains a temporally ordered list of inputs and outputs that occurred during the execution of the private procedure. Since this implementation

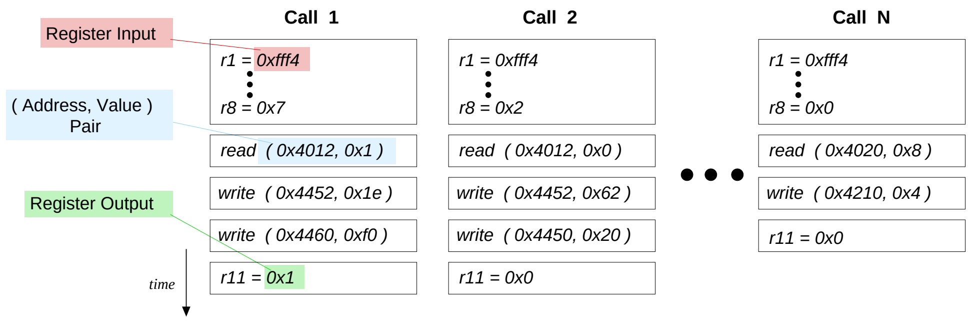


Figure 5-2: Basic private procedure interaction table Ξ.

was based on an AEGIS architecture, we must recognize that values can be passed into a private procedure via registers. Therefore, the start of each column begins with a set of input register values. This includes registers `r3` through `r8` as procedural argument registers, as per the AEGIS Application Binary Interface (ABI). The stack and frame pointer registers (`r1` and `r2`) are also included to account for calls from different points within the application. Each subsequent row within the list is either a read or a write to shared memory, identified by the memory address that is to be read or written to, and the corresponding data value. Finally, the return register (`r11`) is recorded in the last row of the column which is an output of the private procedure.

This ordering of inputs and outputs in the procedure interaction table can now be used by a memoizing adversary to correctly duplicate private procedures which are called during the emulation phase of a Monitor & Swap Attack. An attack that uses an interaction table instead of an input/output relationship pairs table is called a "Temporal Memoization Attack."

5.2 Temporal Memoization

A Temporal Memoization Monitor & Swap Attack constructs an interaction table for every private procedure within a partitioned application and simply "replays" the table whenever a private procedure is called during emulation. Therefore, if a private procedure is called during emulation in the exact same way as it was called during monitoring, the saved interaction table column which corresponds to the monitored call can be used to emulate that procedure's functionality perfectly.

Figure 5-3 depicts the general method by which an interaction table column can be used to emulate a private procedure. When a private partition is called, the initial input values (the argument registers) are compared against the initial input values of every column in the table. All matching columns are set aside as candidate replay columns. Since all application procedures are deterministic functions, the same inputs will lead to the same

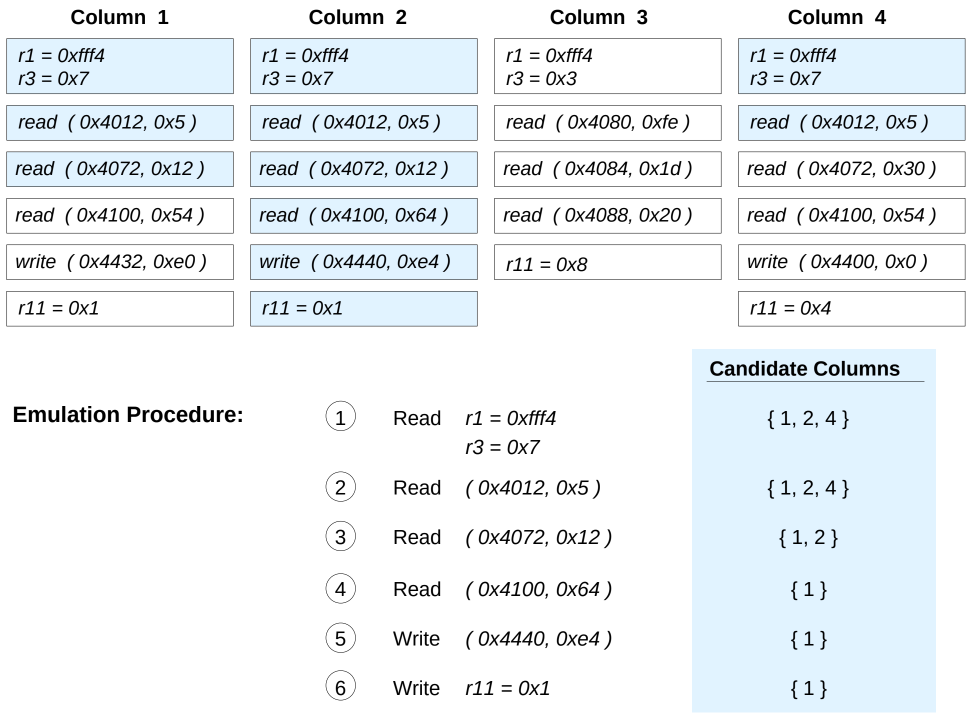


Figure 5-3: General methodology for emulation using Temporal Memoization.

program operation. Therefore the next row of all candidate replay columns will either be the exact same *write* operation, or a *read* operation from the same address. If the next row is a write operation, then that write is performed by the adversary and the subsequent row is inspected in all candidate replay columns. If the next row is a read operation, then the adversary reads that address of public memory and compares the returned value with the values in the next row of all candidate replay columns. Columns that match the value read remain candidates while columns that do not match are removed from the list. This process continues until a row is reached that writes the final output argument (register `r11` on AEGIS).

If the set of candidate replay columns is ever reduced to zero, the attack simply halts with a failure signal. This is because the odds of "guessing" a set of outputs in our configuration are amazingly slim. The chance of this attack correctly guessing the remaining outputs is roughly equivalent to the probability of success of Case (2) of Algorithm 1. Under this particular configuration the probability of success is $P_{call}^{(2)} = \left(\frac{1}{2^{32} \cdot 2^{32}}\right)^s$ since both the output address and value must be chosen for s remaining outputs.

5.2.1 System Setup

To perform a Temporal Memoization Attack, we made a number of modifications to an existing functional AEGIS processor simulator to allow the simulator to play the role of an adversary. (The AEGIS processor, and its functional simulator were originally based on the OpenRISC 1000 project from OpenCores [59].) For simplicity, a new semaphore instruction was added to the AEGIS instruction set that can identify application procedures as either public or private. The simulator is able to execute binary applications compiled for the AEGIS architecture while performing attack-specific tasks when encountering transitions between public and private procedures. An assembly rewriting tool was constructed to automate the insertion of these semaphore instructions.

To begin, the simulator is set to a monitoring mode while an application is executed using only a fraction of its external (I/O) inputs (the application's input workload $\langle \mathbf{\Lambda} \rangle$). After this fraction of the workload has been processed, the simulator halts and writes the contents of the interaction tables to disk. The simulator is then restarted under an emulation mode. After reading in the interaction table from disk, the simulator executes the application on the remainder of the workload. Any private procedures which are encountered at this time are emulated using the interaction table (when possible).

Using a simulator to act as such an adversary mimics a physical attack that directly monitors and controls the physically insecure communication buses leaving the central AEGIS processor. For our experiments we considered a passive adversary who does not change the authentic application in any way and only observes and injects data moving between the processor and external devices such as off-chip memory.

Even though an active adversary can be used in this configuration, we did not perform any experiments which dynamically controlled the inputs to private procedures. This was not investigated since it requires substantial compiler modifications to trap procedure transitions, or a thorough rewrite of application code. Further, a passive adversary performs the exact same attack as an active adversary, only with less information. Any success seen by a passive adversary should directly translate into success for an active adversary.

5.2.2 Compressing the Interaction Table

We found that an important factor effecting the speed (and feasibility) of our Temporal Memoization Attack implementation was the size of the interaction table created during the monitoring phase. If a private procedure is called often during the execution of a partitioned application, its resulting interaction table might become unmanageably large. Not only does this slow the emulation phase, which must search across all columns in the table, but it can also prevent memoization of certain procedures altogether, when the monitoring phase simply uses too much system memory on the host running the simulator.

Although the interaction table contains all the information necessary to emulate a private procedure, it also may contain an abundance of *redundant* information. A Temporal Memoization Attack may only need a subset of the information available in an interaction table.

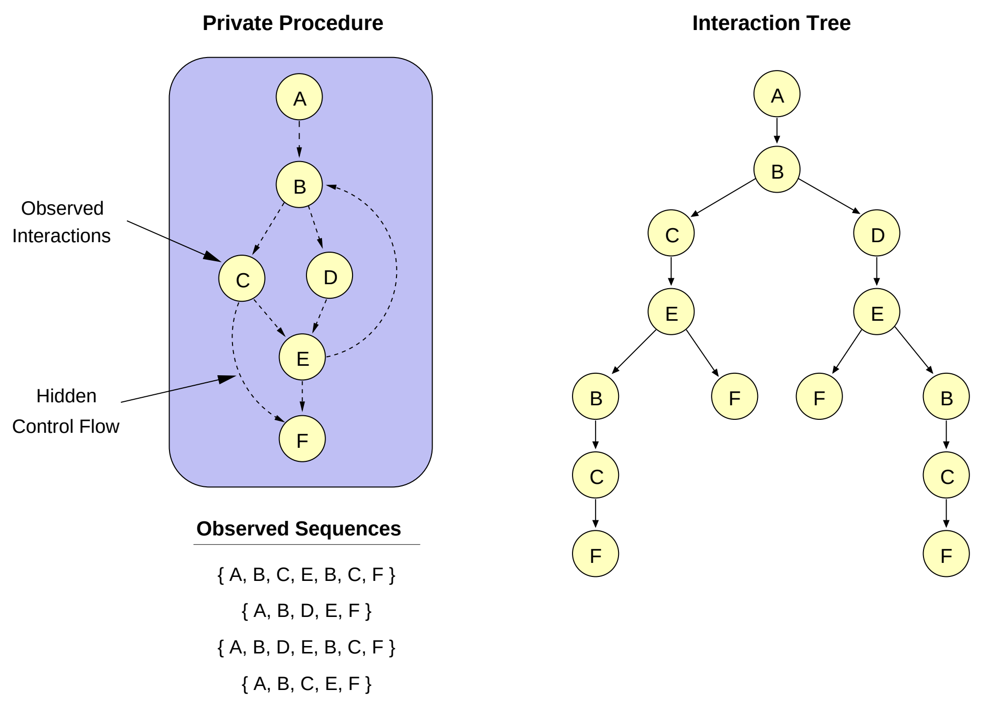


Figure 5-4: Hidden control flow graph represented by a tree.

For this reason, when performing a Temporal Memoization Attack it is usually better to view a private procedure's interaction information as a *tree* instead of a table. Rather than tabulating data, this tree attempts to identify unique execution paths through the hidden control flow graph of the private procedure. Figure 5-4 illustrates this structure. The root of the tree represents the beginning of the private procedure and each branch leaving the root represents one possible execution path. A node along any path in the tree is simply an interaction with public memory that happens to occur during some particular execution path.

To construct our actual interaction tree, we take advantage of a few important properties of deterministic private procedures. First, the absolute ordering of all reads and writes that a private procedure makes does not matter. Only inputs can affect the execution path of a deterministic function so only procedure reads must maintain their absolute order within an interaction tree. This will allow the correct execution path to be replayed during the emulation phase of the attack. Writes must only preserve their ordering relative to other reads, not other writes. To ensure that a write occurs at the correct point in time, an adversary only needs to know the last “read node” of the interaction tree. For example, if a read is followed by three writes and then another read, it only matters that those writes occur *between* the two reads. A counterfeit procedure which adheres to this is guaranteed to be Application Operation Equivalent with its authentic counter-part. (This assumes a sane compiler, one which does not write multiple differing values to an address without reading

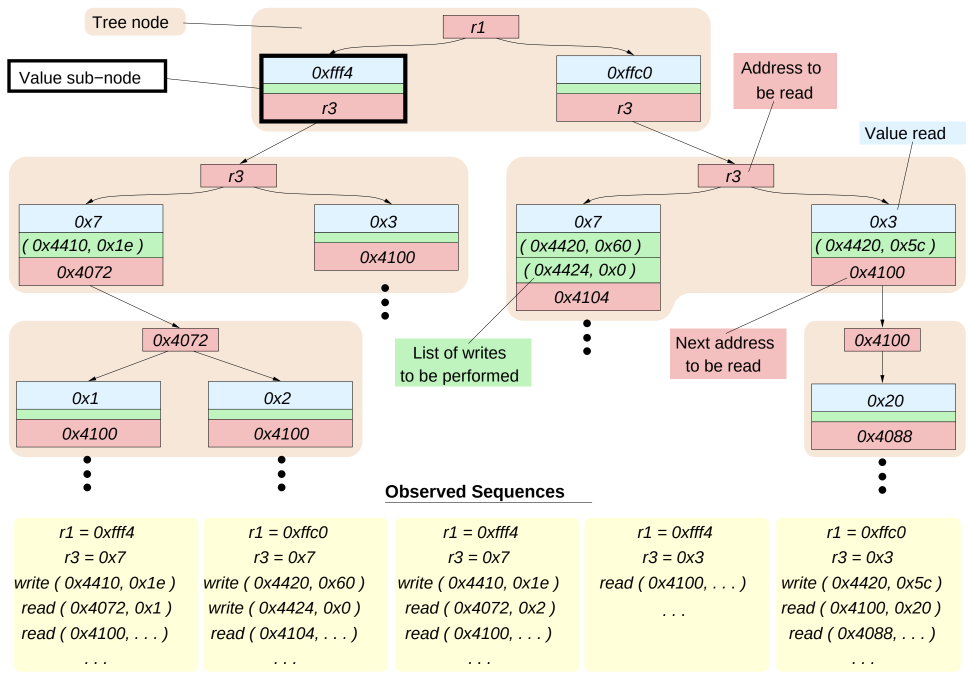


Figure 5-5: Interaction tree for Temporal Memoization.

that address between writes.)

Figure 5-5 shows one possible tree structure that makes use of these properties. Every node of the tree contains an address that identifies which read will be performed next and a number of "value sub-nodes" accounting for every possible value found at that address. These sub-nodes maintain a list of all writes that should be made and a pointer to the next address which will be scanned.

Our final implementation conserved space further by using the same data structures to represent each tree node. Table 5.1 portrays this final structure in tabular form. This

Address	Read Value	Write AV Pairs	Path Number(s)	Next Address
r1	0xfff4	-	0 → 1	r3
	0xffc0	-	0 → 2	r3
r3	0x7	(0x4410, 0x1e)	1	0x4072
	0x7	(0x4420, 0x60) (0x4424, 0x0)	2	0x4104
	0x3	-	1 → 4	0x4100
	0x3	(0x4420, 0x5c)	2 → 5	0x4100
0x4072	0x1	-	1	0x4100
	0x2	-	1 → 3	0x4100
0x4100	0x20	-	5	0x4088
⋮	⋮	⋮	⋮	⋮

Table 5.1: Interaction table used in this implementation of Temporal Memoization.

technique complicates matters since multiple paths can share a single tree node and loops can occur within the interaction tree if the same address is read twice. (Previously, the tree node would simply have been duplicated.) To solve this problem we introduce *path numbers* that are associated with every value sub-node to identify the correct path at any given point during procedure emulation. Generating path numbers during the monitoring phase can be done in a number of ways. However, any effective scheme must pinpoint the initial divergence of execution control paths and identify cases where an address is read multiple times. Numbers are thereby chosen in a manner that allows the emulation phase of the attack to proceed without confusion.

5.2.3 Partially Repeated Workloads

The implementation of a Temporal Memoization Attack can succeed in creating a counterfeit application under a number of different scenarios. One of the most common involves partitioned applications that contain regions of code which perform the same dedicated function over and over (Figure 5-6). This can be the case in countless applications. Some examples include procedures that process data from the initial handshake of a network communication, graphical user interface rendering procedures, and portions of AI computerized opponent algorithms found in games. In fact, it is probable that most applications that do not deal with streamed data (such as video playback) may have small pockets of code that act in this repeated manner.

As one such example, we investigated the "*Parser*" application found in the SPEC CPU2000 [34] benchmark suite. This application begins by processing a set of English dictionary files. It then accepts strings of English phrases and analyzes their grammatical structure. However, instead of English phrases, at times the application can also accept special functional directives in the form of words that begin with the character "`!`." If a partitioning scheme hides one of the procedures that is called when encountering a directive, it may be possible for a Memoization Attack to monitor this procedure and duplicate the directive's functionality. If that procedure is only called in one way for a given directive, it is possible to create a counterfeit application that is Application Operation Equivalent as long as only previously monitored directives are used.

To validate this hypothesis, we designated the `special_command()` C procedure in the `main.c` file of *Parser* as private. We ran the *Parser* application on the simulator in monitoring mode using a text input that issued the "`!echo`" command. This command simply alters a few application variables which cause all text data to be displayed on a screen. The size of the resulting interaction table can be found in Table 5.2. As can be seen, the attack did not consume an exorbitant amount of resources during the monitoring phase. We used this interaction table to run the *Parser* application on the simulator in emulation mode on a number of new inputs, each time also including a `!echo` directive. As expected, the

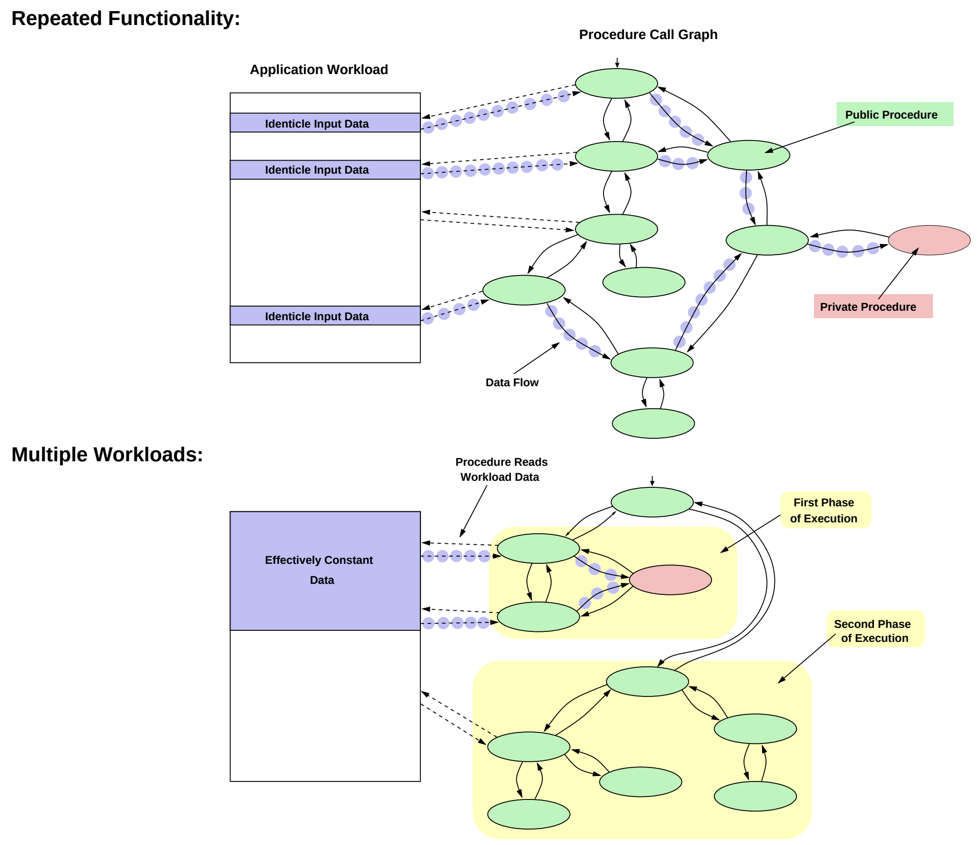


Figure 5-6: Partially repeated workloads.

counterfeit application continued to operate correctly on a number of different text inputs of English phrases before and after the `!echo` call.

Another common scenario involves partitioned applications that operate on multiple workloads, where some of the workloads rarely change (Figure 5-6). Again an example of this is the *Parser* application. This application basically operates in two phases. First it reads in a set of English dictionary files, called the dictionary workload. Second it accepts strings of English phrases for analysis, called the phrase workload. Let us consider a partitioning scheme that privatizes a procedure which is only ever called during the processing of the dictionary workload. If our Temporal Memoization Attack were able to monitor that procedure while the dictionary files were input, we could recreate a counterfeit procedure that duplicates the application's functionality during the dictionary workload phase of execution. Since the counterfeit procedure is never called during the remainder of application execution, this attack creates an AOE counterfeit application for a specific dictionary file (which is unlikely to change often, if at all).

To again demonstrate this feat, we made the `is_equal()` procedure private in the `read-dict.c` file of *Parser*. This is a very simple function, but is an example of one

Metric	*Parser* `special_command()`	*Parser* `is_equal()`
Total number of nodes in tree/table	283	5
Total number of value sub-nodes in tree/table	545	76,891
Size on disk (in Bytes)	26,972	3,042,968
Equivalent maximum depth of interaction tree	743	5

Table 5.2: Size of memoized private procedures.

of many small functions that only get called during the phase which reads in the dictionary workload. The *Parser* application was then monitored by the simulator while executing using the tiny English phrase reference input "`smred.in`," as well as the standard SPEC reference dictionary input. The `smred.in` file is a small replacement for the standard SPEC workload, taken from MinneSPEC [44]. It only contains one English phrase to be parsed, opposed to the $7,760$ phrases found in the standard SPEC workload. The size of the resulting interaction table is again reported in Table 5.2. Using this interaction table, we ran the *Parser* application on the simulator while in emulation mode. Previously unseen English phrase file inputs were used, including the `mdred.in` and `lgred.in` input files found in MinneSPEC. As expected, the counterfeit application appeared to operate correctly no matter what set of English phrases we provided it.

5.2.4 Composite Workloads

It is also possible for a Temporal Memoization Attack to succeed in creating a counterfeit application even when the external input workload does not contain regions of congruence. Depending upon the application, it is possible for different workloads to still cause constituent private procedures to be called using the same set of inputs. That is, the set of input/output relationship pairs from private procedure calls made while executing on some fixed workload can be a strict subset of the set of input/output relationship pairs from the private procedure calls made when the application is run on a set of completely independent workloads. If a private procedure exhibits this property, an adversary can succeed in a Memoization Attack by simply observing the authentic application executing on *any* set of workloads over a long period of time.

When an application designer decides to partition an application (privatizing some procedures), it may be easy to identify procedures that are only ever used by repeated workloads, and are therefore susceptible to emulation attacks. However, intuition alone cannot identify procedures that are susceptible to emulation attacks given a collection of independent workloads. Here we show how frequently this characteristic presents itself in real

Gzip[4] procedure (Lines of assembly)	Percentage of correct procedure calls while emulating `ref.log` after observing workload `ref.*`			
	`random`	+`graphic`	+`program`	+`source`
`bi_reverse` (11)	38% (681/1797)	76% (1362/1797)	84% (1518/1797)	97% (1741/1797)
`huft_build` (438)	0% (0/27)	0% (0/27)	0% (0/27)	0% (0/27)

Parser procedure (Lines of assembly)	Percentage of correct procedure calls after observing workload `lgred.in`	
	emulating `mdred.in`	emulating `smred.in`
`contains_one` (123)	33% (1136/3485)	0% (0/71)
`set_has_fat_down` (58)	0% (0/61)	0% (0/1)

Table 5.3: Success of Temporal Memoization Attack on real applications.

applications. Table 5.3 displays the results of a Temporal Memoization Attack when using a composite set of workloads to attempt to emulate procedures from the *Gzip* and *Parser* applications found in the SPEC CPU2000 benchmark suite.

In the attack of the *Gzip* application, the workload `ref.log` is emulated after observing the execution of *Gzip* on the `ref.random` workload, the `ref.random` *and* `ref.graphic` workloads, and so on (using a 4MB chunk size). All five of these workloads are completely independent data files meant to be represent different types of compressible information. Therefore there should be almost no redundancy between the `ref.log` and any of the other reference workloads. As we can see, the `bi_reverse()` procedure can be emulated almost entirely correctly when running on the `ref.log` workload if the other four reference workloads have already been observed. Of the $1,797$ calls made to `bi_reverse()` during the processing of `ref.log`, $1,741$ of the calls contains the *exact* same inputs as had been seen with the first four workloads. Given this, it seems reasonable to conclude that the `bi_reverse()` procedure is a poor choice for privatization by an application designer.

In the attack of the *Parser* application, the `mdred.in` and `smred.in` workloads (again from MinneSPEC) are emulated after observing the execution of the application using the `lgred.in` workload. Again, the `mdred.in` and `smred.in` workloads are completely independent input data files from the `lgred.in` workload. Although it appears that the procedures listed cannot be emulated completely correctly given only the `lgred.in` workload, there still appears to be a large number of duplicated procedure calls given this small workload. This leaves open the possibility that some procedures in the *Parser* application can be emulated correctly if suitable observed and emulated workload sets are encountered. This limited analysis of the *Gzip* and *Parser* applications makes it clear that real application procedures can be emulated by a Memoization Attack, even on completely different workloads.

5.3 When Temporal Memoization Fails

Despite what we have shown, the success of a Temporal Memoization Attack will always depend on the number of input values it observes during the monitoring phase of an attack.

This attack will fail during the emulation phase the moment an input is provided to a private procedure that had not been encountered during monitoring. However, it may be possible to use properties common to any application procedure to probabilistically associate previously unseen inputs with existing inputs and interaction table columns. This can effectively "expand" the set of inputs that can be handled by a Temporal Memoization Attack.

Here we discuss some of the methods of associating new inputs with existing interaction table columns, highlighting realistic procedural property assumptions. These methods are only utilized when a Temporal Memoization Attack reads an input during emulation that is not already in the interaction tree. Note that, formally, any such expansion is simply a technique used to "guess" the output of a private procedure given a set of inputs that have not been seen before (Case 2 of Algorithm 1). Although none of these methods were implemented, they are included here because they represent a number of "fair" assumptions a realistic adversary may make when attempting to implement a Temporal Memoization Attack of his own. Therefore these serve as practical methods that offer some utility in determining the output distribution of a private procedure.

5.3.1 Read Value Distance

One of the simplest ways to decide what to do when a new input value is observed is to continue the emulation process using a value within the interaction tree that is "closest" to the new value. To do this an adversary must define some function $D(x, y, \sigma)$ that returns the distance between inputs x and y given possible external knowledge σ. This can be something simple such as the cardinal or Hamming distance (which do not use σ), or it can be more complex, maintaining a history of prior input values and returning a distance value which is normalized against some existing maximum likelihood gap estimator.

When an adversary encounters a previously unseen input during emulation, he simply applies $D(\cdot)$ to the input value and every existing value in that node of the interaction tree. The new input value is then ignored, and the existing value that returned the smallest distance is used in its place. That value sub-node is then used to determine what writes must be applied and what address to read next.

5.3.2 Address Path History

Ideally, an adversary would like to associate every node within an interaction tree with some node in the true (hidden) control flow graph (CFG). If a procedure's control flow graph is known, an adversary can always choose the next input read address correctly. Determining what writes should be made is also simplified by this knowledge.

Although it is impossible to discern this control flow graph, an adversary can provisionally construct a CFG of his own based on prior sequences of observed reads. One

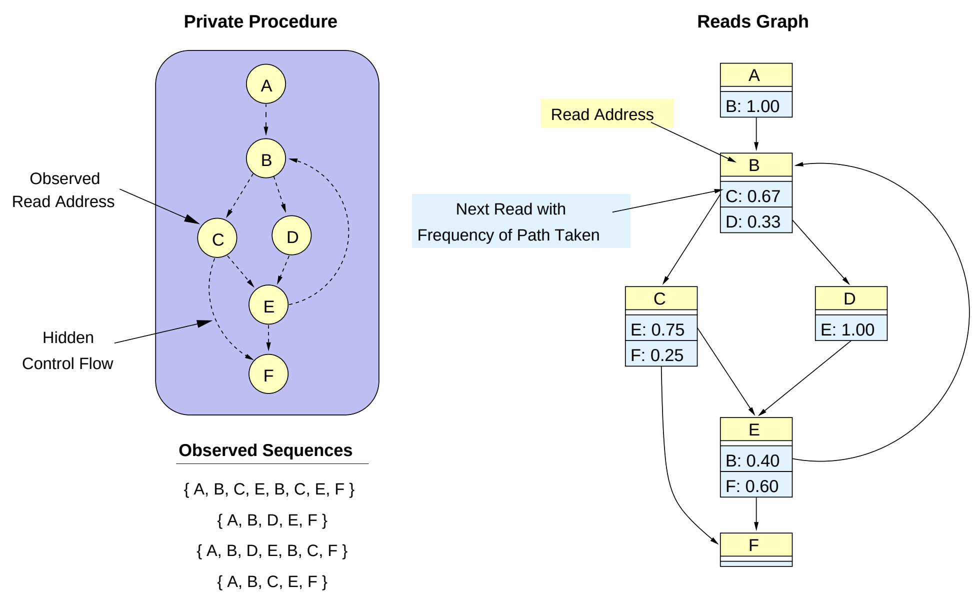


Figure 5-7: Reads Graph attempting to match hidden control flow graph.

straightforward technique for this builds a "*Reads Graph*" during the monitor phase of an attack which is separate from the interaction tree. As can be seen in Figure 5-7, the reads graph contains a node for every procedural read. Each node keeps a list of all possible next read addresses, along with the percentage of times each next address was the taken chosen path.

While an adversary emulates a private procedure by stepping through nodes in the procedure's interaction tree, he concurrently steps through nodes in the reads graph. When a new input is seen, the adversary finds the set of value sub-nodes within the interaction tree that have the same next read address as the most likely next read address in the reads graph. The adversary can then use a different method, such as value distance, to decide which of these value sub-nodes to use to determine what writes to perform.

It is possible to extend this method by conditioning each percentage value in a reads graph node according to what reads have previously been seen. This will help disambiguate procedural loops. Further, a history containing a list of all reads in the last N calls to a private procedure can help identify larger application loops which typically make calls to a procedure over and over using a fixed number of inputs.

Chapter 6

Indicators of Insecurity

It would be most comforting if a single test existed that could be applied to a private procedure of an application which identifies the "amount" of secrecy inherent in the procedure. Furthermore, this secrecy score must remain constant for any possible usage of the procedure that different applications may have, and must specify whether or not the procedure can be sufficiently emulated to allow these applications to run correctly. Any such test would have to include information theoretic assessments of the entropy and complexity [71] of both the private procedure and the application, as well as an accurate model of the "learning ability" [73, 74] of all possible adversaries. Even when only dealing with adversaries who are aware of input/output relationship pairs, a test like this is practically infeasible to construct in a way that applies to a general set of applications.

Consequently, this work proposes "indicators of insecurity" that *speculate* upon the likelihood that a private procedure can be emulated in a partitioned application. That is, these indicator tests identify vulnerabilities that may invalidate any assumed functional secrecy of a private procedure. If a partitioned application "passes" an entire set of these tests, then a designer can have some degree of confidence that the private regions of an application cannot be emulated when used by that particular application. This method of identifying negative results is a common technique used when dealing with problems that do not have a clear positive indicator. For example, random number generators are tested against suites of statistical tests that can only identify "non-random" sequences [36, 42, 47]. This is because the concept of an "is-random" test is murky at best.

Since the security focus of this work is Application Operation Equivalence, the assessments proposed examine a private procedure's interaction with the whole application. The internal functionality of a private procedure is only of indirect interest if that functionality effects the procedure's external interactions. Put another way, the private procedures here are always considered "black-boxes."

6.1 Empirical Correlations

To determine the effectiveness of a Memoization Attack on a particular partitioned application, it seems simple enough to *run* a Memoization Attack and see if it succeeds. Performing an attack to verify the secrecy of private procedures can merely be one last step of the application design process. Unfortunately a Memoization Attack can require a large amount of time and resources dependent on the size of the input workload that is used for the monitoring phase of the attack. Further, an adversary may only need to attack a single private procedure while an application designer must check every private procedure. This creates an imbalance between the amount of computation that must be done to attack versus the amount of computation which must be done to defend.

Therefore, it may be easier for an application designer to look at a set of simple, efficient tests that examine the interaction between private and public procedures and attempt to discover whether a private procedure can be emulated or not. If these tests discover weaknesses which correlate with a Memoization Monitor & Swap Attack, then it may be possible to use these tests to efficiently scan a partitioned application for easily emulated private procedures.

6.2 Input Saturation

The number of inputs a private procedure requires directly impacts how difficult it is for an adversary to emulate a procedure as well as the practical size of the input/output relationship pairs table $\mathbf{\Pi}$ (and correspondingly the size of the interaction table $\mathbf{\Xi}$ in a Temporal Memoization Attack). Every unique input value, be it a register argument or a memory read, increases the number of elements within the inputs vector $\boldsymbol{\lambda}$. Intuitively, the more elements within the inputs vector, the harder it is for an adversary to emulate the procedure since there are a greater number of possible values that an adversary must observe during the monitoring phase. For example, every read encountered during a Temporal Memoization Attack can possibly create a new branch in the interaction tree. During emulation, every branch point is basically one more adversarial decision that must be made.

6.2.1 Individual Input Counting

Each input to a private procedure is an entire set of values, indexed by their addresses, defined by the vector $\boldsymbol{\lambda}$. If a procedure takes q argument inputs (addresses), and each input can take m unique values, then the total number of possible input sets for that procedure is m^q. Even though only $q \cdot m$ unique Address/Value (AV) pairs were used, the procedure can function differently for every different combination.

An adversary executing a Memoization Attack must keep track of all m^q input sets seen during the monitoring phase. However, application procedures are often more affected by the value of an individual argument, rather than the entire combination of argument values. For example, conditional checks tend to focus on single argument values to determine control flow and to decide which arguments to use later. Further, since there might only be a few possible control flow paths within a procedure it makes sense that the number of unique AV pairs used by a procedure correlates with the number of possible input sets a procedure uses. Given this belief, it follows that a high number of unique AV pairs observed corresponds to a low likelihood that a private procedure can be emulated. Therefore, an application designer can simply count the number of unique AV pairs when determining if a private procedure is likely secure. Counting the number of unique AV pairs is a much simpler computational feat since there are no greater than $q \cdot m$ pairs.

When a private procedure first begins its execution, the number of unique input AV pairs that are observed as inputs can be plotted on a graph as a function of time or the number of procedure calls. We have seen in experiments that at the beginning of execution the number of unique inputs observed always increases greatly over a relatively few number of procedure calls. This makes sense since proficient designers only create procedures as a means to abstract away complicated application behavior. However, as an application continues to process some workload, and more calls to a private procedure are made, some procedures continue to exhibit a drastic increase in the number of unique input values while others do not. These latter procedures tend to exhibit a very slow and diminishing rate of increase in the number of unique inputs seen as more calls are made. These procedures appear to be approaching some kind of maximum number of input AV pairs and are therefore called "*input saturating procedures.*" Since this implies that many or most of the possible inputs can quickly be discovered by an adversary, input saturation is an important concern for partitioned application designers. Further, the rate at which a private procedure requires new input sets directly affects the length of time ω for which an adversary can construct a T-AOE application.

Unfortunately, whether or not a private procedure is an input saturating procedure is an experimentally and intuitively derived answer. This is because, generally, the total number of possible inputs to a procedure cannot be determined statically at a compile-time (because of the inter-procedural aliasing problem [16, 72]), and it is intractable to enumerate every possible set of inputs to determine which sets are useful to a procedure and which are not. It may be possible to use approximating algorithms to estimate the values that are read by a procedure, but the easiest way for an application designer to determine the number of unique input AV pairs a procedure reads is to simply count the number of pairs while running the application on some workload using either a functional simulator or a binary instrumentation tool [62]. With this, saturation can only be hypothesized by a designer by "eyeballing" a "cumulative input density function," which is a graph that plots the number

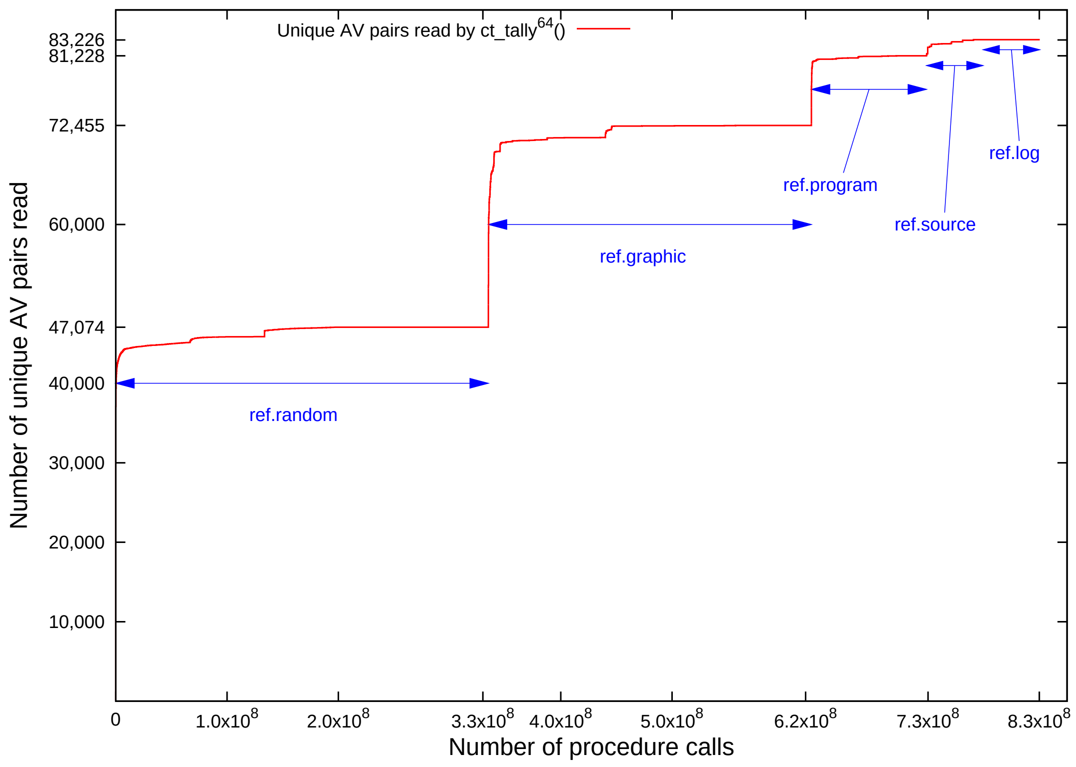


Figure 6-1: Cumulative input density function of `ct_tally`^{64}`()` from *Gzip*.

of unique inputs observed as a function of the number of times a procedure is called. Briefly put, input saturating procedures tend to exhibit plots that flatten as more procedure calls are made.

Figure 6-1 is a graph of the number of unique inputs seen as a function of the number of calls to the `ct_tally()` procedure in the SPEC CPU2000 [34] benchmark *Gzip*. For this experiment, *Gzip* was run on five different reference workloads taken from SPEC CPU2000, where each workload was expanded to a 64MB chunk size before compression. (The workloads were ordered `ref.random`, `ref.graphic`, `ref.program`, `ref.source`, `ref.log`.) Notice that the rate of increase in the number of unique inputs decreases as more workloads are applied (each workload producing the four noticeable bumps). In fact, one can see that the `ref.log` workload did not cause *any* new input AV pairs to be seen (hence no fifth bump). Given this particular plot, an application designer can assume that the `ct_tally()` procedure is likely input saturating.

More formally, input saturation can be quantified in a number of ways. First, determining the average percentage increase of the number of unique input AV pairs from call to call can give an indication of whether a procedure is input saturating. Not only that, but if we also assume that the number of unique input AV pairs observed correlates with the number of input sets observed, then the average percentage increase can give an estimate of how many procedure calls are expected to take place before a new input set is observed. This is exactly ω in the formulation of T-AOE if the named procedure is used as a private procedure in an authentic application. This percentage increase is called the "*average input delta*," (*Avg.* $I\Delta\%$). The smaller the average input delta, the more likely it is that the procedure is input saturating.

Another way to quantify the input saturation of one private procedure is by relative comparison with other procedures. This is useful since it is often sufficient for an application designer to simply know which procedures are less input saturating than others. For this, we define a "*saturation weight*" (SW) for any procedure that has been monitored over the course of N procedural calls. If the function $w(c)$ represents the number of unique input AV pairs given the number of calls c, then the saturation weight is simply the integral of $w(c)$ from 0 to N, normalized against the maximum value of $w(c)$ and N. Therefore,

$$SW = \frac{1}{Nw(N)} \int_0^N w(c)\, dc.$$

6.2.2 Real-World Saturation Rates

To get a feeling for the prevalence of input saturating procedures in real applications, we analyzed the input values of all of the procedures that make up the *Gzip* benchmark. As before, *Gzip* was run on the five SPEC CPU2000 reference workloads, only this time using a chunk size of 3MB (to permit analysis of the entire application).

Table 6.1 shows the increase in the number of inputs of five selected procedures, as well as the 64MB chunk `ct_tally()` procedure from Figure 6-1. Figure 6-2 displays the cumulative input density function of these five procedures. Since different procedures are called a different number of times during the execution of an application, this graph normalizes the number of calls with the maximum number of times each procedure was called (N). Similarly, the number of unique input AV pairs is normalized with the maximum number of unique input AV pairs each procedure ever sees ($w(N)$).

From this table and figure a designer might infer that the `ct_tally()` and `bi_reverse()`

Procedure	Unique inputs seen after execution on workload `ref.*`					*Avg.* $I\Delta\%$	SW
	`random`	+`graphic`	+`program`	+`source`	+`log`		
`ct_tally`^{64}	47,074	72,455	81,228	83,226	83,226	9.7x10^{-9}	0.77
`ct_tally`^{3}	2,304	2,550	2,768	2,836	2,837	6.9x10^{-7}	0.87
`bi_reverse`^{3}	569	580	580	580	581	6.3x10^{-5}	0.99
`huft_build`^{3}	0	2,500	3,170	3,510	3,586	7.4x10^{-3}	0.72
`build_tree`^{3}	11,873	23,611	29,945	32,103	32,672	5.9x10^{-3}	0.51
`longest_match`^{3}	4.78 M	8.33 M	10.13 M	11.19 M	11.61 M	2.7x10^{-6}	0.51

Table 6.1: Rate of input saturation for five *Gzip* procedures.

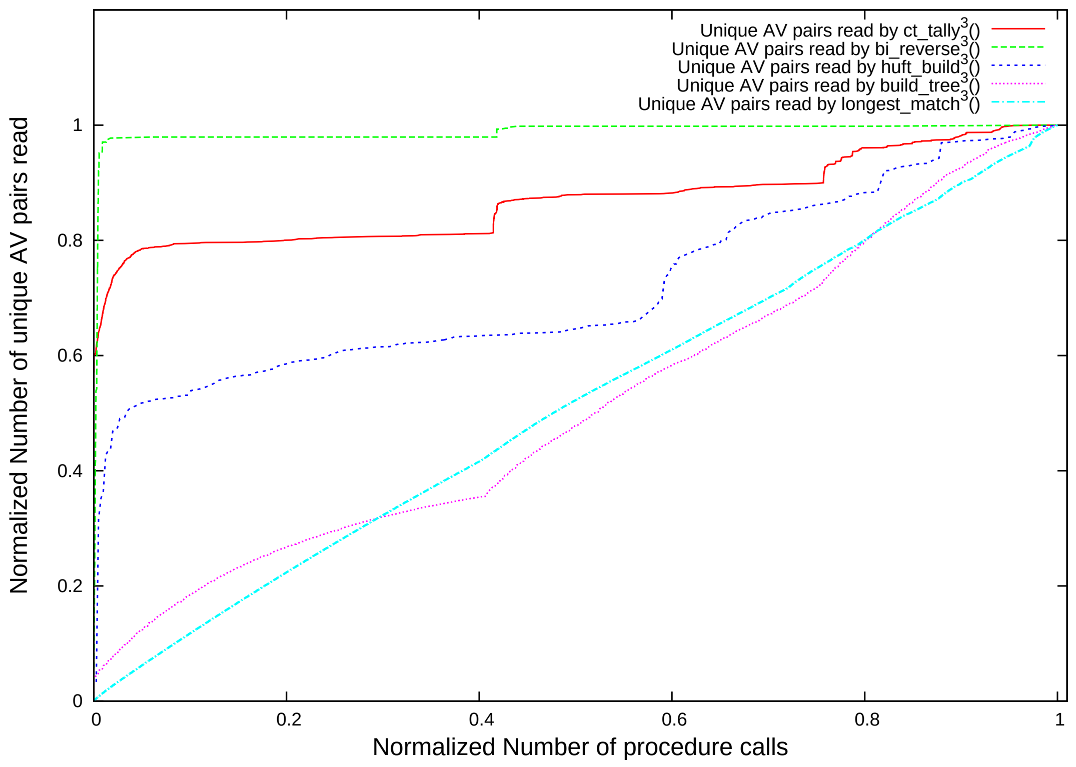


Figure 6-2: Cumulative input density functions from *Gzip*.

procedures are probably input saturating while the `build_tree()` and `longest_match()` procedures are probably not. It is less clear if the `huft_build()` procedure is input saturating since its cumulative input density function seems to continue to grow steadily, albeit less quickly than that of `build_tree()` or `longest_match()`. When comparing procedures, one can see that non-input saturating procedures tend to have *SW* values around 0.5 while input saturating procedures tend to have values much closer to 1.0. The average input delta value of a procedure cannot be readily used for comparison, but instead gives a hint at the T-AOE ω value of a counterfeit application that can be constructed by an adversary who observes the same inputs and outputs.

It is interesting to note the drastic difference between the number of input AV pairs seen by the 64MB chunk size `ct_tally()` versus the number of input AV pairs seen by the 3MB chunk size `ct_tally()`. Even though the `ct_tally()` is presumed to be input saturating, these two experiments can result in different saturation levels because the *Gzip* application is being used in a somewhat different way, effectively on different workloads. During the processing of data, the 64MB chunk size *Gzip* application calls the `ct_tally()` procedure nearly 120 times more often than the 3MB chunk size *Gzip* application (7M versus 834M). Given the observations of the 64MB chunk size application, it can be seen that the *SW* value of `ct_tally()` is not quite as high as it appears after observations of the 3MB chunk size application. This emphasizes the need for an application designer to perform exhaustive experiments to determine if a procedure is secure, and underscores the fact that a designer cannot definitively ascertain the maximum number of unique input AV pairs a procedure uses.

Further making this point is the variation in the number of calls made to any procedure given different workloads (most drastically seen with the `huft_build()` procedure). Although all five workloads compress 3MB chunks of data, the `ref.random` and `ref.graphic` workloads tend to make nearly 15 times the number of calls to these procedures as are made for the `ref.source` and `ref.log` workloads. This is noticeable in the inconsistent bump appearances in Figure 6-1 and Figure 6-2.

6.3 Data Egress

Although examination of the inputs of a private procedure can give strong indications of whether a procedure can be emulated or not, it is the outputs of a procedure that an adversary must actually recreate during the emulation phase of a Monitor & Swap Attack. Therefore, inspecting the usage and the number of unique outputs may be a better metric to identify emulation susceptibility.

6.3.1 Output Counting

Generally, one might expect that private procedures that generate more output values are harder for an adversary to emulate. During procedure emulation, it is clear that any abstract Memoization Attack (for example, Algorithm 1) requires an independent guess for each and every output when a set of inputs are encountered that had not been observed during the monitoring phase. Akin to individual input counting, if there are m possible unique AV pairs that an individual output can produce, and there are q outputs, then there are m^q possible sets of outputs a procedure can generate. This large number can effect an adversary's chances of guessing outputs during a Memoization Attack, however, it is likely the case that only a portion of these sets of outputs can ever be created for one specific procedure. An application designer would ideally like to select procedures for privatization that have a considerable number of possible sets of outputs, thereby decreasing an adversary's chances of success when using a Memoization Attack.

While an adversary performing a Memoization Attack must remember every complete set of outputs he observes, it may be sufficient for an application designer to simply count the number of unique output AV pairs as an indication of the number of sets of outputs. This follows from the same argument found in input saturation. The number of unique output AV pairs usually correlates with the total number of unique sets of outputs because of a relatively few number of possible control flows paths within any procedure. Again, only counting the number of unique output AV pairs is a far simpler computation task since only a maximum of $q \cdot m$ pairs will ever be observed.

An application designer would therefore want to know the "*output egress count*" (the total number of unique writes performed) of any procedure under consideration for privatization. However, since it is possible for the outputs of a procedure to be incorrect while the application as a whole operates correctly, a simple count of the number of output pairs only captures a part of the problem. As a rudimentary means of capturing the importance of a procedure's outputs, the "*output egress weight*" metric is also presented here. The goal of this weight is to assign a single value to any procedure that gives an indication of how easy it is for an adversary to guess a set of outputs and still allow the application as a whole to continue to run correctly. Again, determining this metric is meant to consume very few computational resources so that it can be executed quickly on even the largest of applications. Therefore it is only a vague approximation of the likelihood of an adversary emulating a given procedure.

The output egress weight of a private procedure is any function $\Phi(\cdot)$ that takes into account the outputs of a procedure and the manner in which the outputs are used. For demonstrative purposes, we present here a simple $\Phi(\cdot)$ function that is a combination of the number of unique AV pairs which are output by a procedure and the number of public procedures that use these outputs. Recognizing that a private procedure can only impact the outputs of the entire application if its own outputs are passed along by other procedures, we define this example $\Phi(\cdot)$ function to be

$$\Phi(\boldsymbol{\eta}), \quad = \sum_{\forall(\iota_i,\kappa_i)\in\boldsymbol{\eta}} \frac{\kappa_i}{\iota_i}.$$

Here $\boldsymbol{\eta}$ is a set of pairs (ι, κ), where ι is the number of unique output AV pairs written by a private procedure and read by a public procedure, and κ is of the total number unique output AV pairs of that public procedure. For example, if five public procedures use the outputs of a private procedure as inputs, then $|\boldsymbol{\eta}| = 5$. The fraction $\frac{\kappa_i}{\iota_i}$ is used to give a hint of the possible impact of any private procedure output on the outputs of the corresponding public procedure. A larger value of $\frac{\kappa_i}{\iota_i}$ can imply that a private procedure's output values are important while a small value of $\frac{\kappa_i}{\iota_i}$ might mean that many of the private procedure's output values effectively produce the same result.

As an example, Figure 6-3 shows the number of unique output AV pairs that were observed from the `inflate_codes()` procedure in the SPEC CPU2000 *Gzip* application when run on the five reference workloads provided by SPEC CPU2000 (using a 3MB chunk size). The private procedure's outputs were identified by tracking public procedure reads of AV pairs that the private procedure had most recently written. Notice that the sum of all the edges is greater than the total number of writes performed by the private procedure

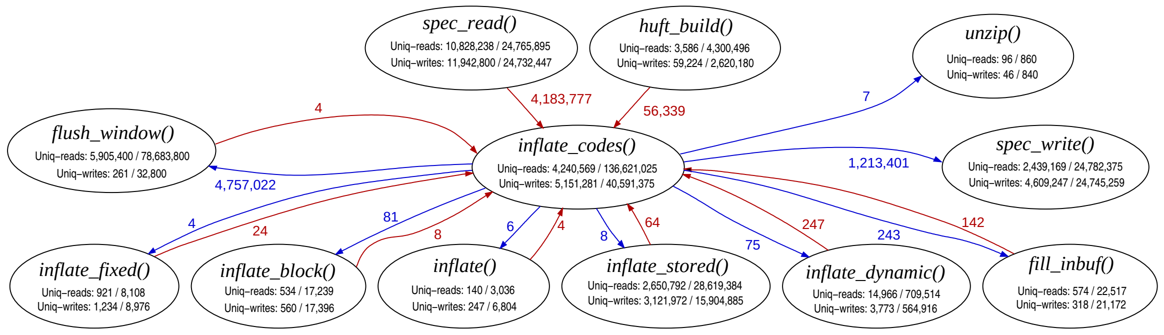

spec_read()
Uniq–reads: 10,828,238 / 24,765,895
Uniq–writes: 11,942,800 / 24,732,447
huft_build()
Uniq–reads: 3,586 / 4,300,496
Uniq–writes: 59,224 / 2,620,180
unzip()
Uniq–reads: 96 / 860
Uniq–writes: 46 / 840
flush_window()
Uniq–reads: 5,905,400 / 78,683,800
Uniq–writes: 261 / 32,800
inflate_codes()
Uniq–reads: 4,240,569 / 136,621,025
Uniq–writes: 5,151,281 / 40,591,375
spec_write()
Uniq–reads: 2,439,169 / 24,782,375
Uniq–writes: 4,609,247 / 24,745,259
inflate_fixed()
Uniq–reads: 921 / 8,108
Uniq–writes: 1,234 / 8,976
inflate_block()
Uniq–reads: 534 / 17,239
Uniq–writes: 560 / 17,396
inflate()
Uniq–reads: 140 / 3,036
Uniq–writes: 247 / 6,804
inflate_stored()
Uniq–reads: 2,650,792 / 28,619,384
Uniq–writes: 3,121,972 / 15,904,885
inflate_dynamic()
Uniq–reads: 14,966 / 709,514
Uniq–writes: 3,773 / 564,916
fill_inbuf()
Uniq–reads: 574 / 22,517
Uniq–writes: 318 / 21,172
4
4,183,777
56,339
7
4,757,022
1,213,401
4
24
81
8
6
4
8
64
75
247
243
142

Figure 6-3: Unique outputs of the `inflate_codes()` procedure in *Gzip*.

(labeled within the node as "Uniq-Writes"). This is because multiple public procedures can read the same value that the private procedure wrote. From this graph the output egress weight of the `inflate_codes()` procedure can be determined to be

$$\begin{aligned}\Phi(\cdot) &= \frac{4,757,022}{261} + \frac{1,234}{4} + \frac{560}{81} + \frac{247}{6} + \frac{3,121,972}{8} + \frac{3,773}{75} + \frac{318}{243} + \frac{4,609,247}{1,213,401} \\ &= 390,657.\end{aligned}$$

Similar to input saturation, the easiest way to ascertain the output egress weight of a procedure is by monitoring a procedure while the application is executed on a set of workloads. Static analysis would be as problematic as it is for input saturation.

6.3.2 Real-World Output Egress Weights and Counts

We again assert that procedures that exhibit a relatively low output egress weight or count are suspect and poor candidates for privatization. This is based on the assumption that a procedure with a fewer number of different outputs and a fewer number of public procedures that actually use the outputs are probably easier for an adversary to emulate. Again, we examined a set of procedures in the *Gzip* application to gain a sense of typical output egress weights and counts.

Table 6.2 looks at six procedures from the *Gzip* application and counts the total number of unique reads performed by each procedure and the total number of unique writes that each procedures makes (the output egress count). The number of procedures that receive data from each procedure is also listed along with the $\Phi(\cdot)$ weight. These statistics were generated by executing *Gzip* on all five of the SPEC CPU2000 *Gzip* application workloads using a 3MB chunk size. Again, the total number of reads and writes does not necessarily coincide with the summation of all incoming or outgoing edges as shown in the graph in Figure 6-3.

The tables shows that the `inflate_codes()` and `ct_tally()` procedures produce a lot

Procedure	Total unique reads	Output egress count	Recipient procedures	$\Phi(\cdot)$ weight
`inflate_codes`	4,240,569	5,151,281	9	390,657
`ct_tally`	2,837	4,214,758	4	1,343,144
`bi_reverse`	581	259	2	93
`huft_build`	3,586	59,224	4	96
`build_tree`	32,672	21,000	4	2
`longest_match`	11,610,835	515	1	13,010

Table 6.2: Output egress weights and counts for six *Gzip* procedures.

of unique output AV pairs that are read by many different procedures which in turn produce even more outputs. The high $\Phi(\cdot)$ weight values indicate that it is possible for the outputs of these two procedures to affect an even larger set of outputs within the entire application. If this trend continues, it is likely that the final outputs of the application may be greatly affected by the outputs of these two procedures. Under the assumption that the number of unique output AV pairs correlates with the number of sets of outputs, this implies that these procedures may be difficult for an adversary to emulate in a Monitor & Swap Attack since the adversary must recreate a large number of outputs. Further, most of those outputs must be emulated correctly since they probably have a strong impact on the final application outputs.

The `bi_reverse()`, `huft_build()`, `build_tree()`, and `longest_match()` procedures only produce a limited number of unique output AV pairs and these outputs are passed to procedures that, in turn, do not produce that many more unique outputs. Given this low $\Phi(\cdot)$ value, there is a possibility that the outputs of these procedures do not greatly affect the final outputs of the application as a whole (although this is only an estimation). Because of this, and more importantly because of the low output egress count, these procedures may not be the best candidates for privatization. Again assuming the correlation between unique output AV pair counts and the number of sets of outputs, an adversary performing a Memoization Attack might find it easy to reproduce the outputs of these procedures since it appears that only a small number of unique sets of outputs can ever be generated. Further, it might be possible for an adversary to incorrectly emulate an output, and yet have the application as a whole continue to run correctly.

6.4 Application Designer Use of Indicators

All told, this chapter highlights the importance of an application designer analyzing individual procedures before he entrusts them to be private procedures. Whenever possible the designer should use as many "tests" as possible to determine if a particular procedure is susceptible to attack, namely the Memoization Monitor & Swap Attack.

Input saturation and output egress weights and counts are three such tests that an application designer can use to attempt to identify procedures that an adversary might be able to emulate using a Memoization Attack. However, it is crucial that the designer use these tests in tandem. As we can see, the set of "safe" procedures that the input saturation test determines does not perfectly overlap with the set of safe procedures which the output egress weights and counts tests determines. For example, the `ct_tally()` procedure appears to produce enough unique output AV pairs and has a high enough $\Phi(\cdot)$ weight to warrant consideration for privatization. However, this procedure clearly appears to be input saturating, as is seen in Figure 6-2.

In the end, the total number of unique sets of inputs and outputs of a procedure cannot be statically determined so all of these tests are merely estimates based on the execution of an application on a specific workload. To this end, any indicator which attempts to identify procedures that are susceptible to attack can only ever be an estimation since the types and models of different attacks are nearly endless.

Chapter 7

Partitioning Methodologies

As previously discussed, one of the major goals of an application designer is to control who is able and who is unable to use his application, that is, software licensing. The AEGIS architecture used in this work realizes this goal by encrypting application code so that it can only be executed on a single processor. The physically secure architecture itself ensures this, however, it relies on the underlying encryption technique to protect the privacy of the application code. If an adversary is able to discover the contents of an encrypted application, he can circumvent AEGIS's software licensing scheme by re-encrypting the exposed application code for whatever processor he likes.

Common sense suggests that encrypting the entire contents of an application with a provenly secure encryption algorithm affords a high level of code privacy. However, an application designer implementing software for the AEGIS architecture would prefer to partition his code so that only a few small procedures are encrypted while the majority of the application remains unencrypted. This improves the performance of the application as a whole, since encrypted code executes more slowly, while still binding the application to a single consumer system containing the only processor that is able to unencrypt these procedures.

Unfortunately, previous chapters show that partitioned applications can indeed reveal enough information to render encryption irrelevant. Because of this, ensuring the privacy of partitioned regions requires considerable attention when making the decision of what to privatize, else a private procedure might be chosen that can be easily attacked. Assessments can be made that identify a potentially insecure private procedure, however, there is no upper boundary test which can claim that a private procedure is "secret enough." Therefore an application designer would like to privatize the "most secret" procedure possible.

As far as performance goes, minimizing the execution time of an application requires a minimization of the amount of code placed in private procedures. However, it is often the case that maximizing the secrecy of a private procedure requires a maximization of the amount of code within the private procedure. Therefore an application designer is faced with a fundamental tradeoff between performance and security when deciding which regions of an application to hide and which regions to leave public.

Furthermore, the secrecy of a private procedure is not the only concern faced when ensuring a robust software licensing scheme. The choice of procedure matters too. The procedure to be privatized must be a commonly used, crucial component of the application as a whole, otherwise an adversary can simply remove the private procedure and retain the original functionality given *most* workloads.

7.1 Essential Characteristics

To begin, let us discuss some of the essential application behavior characteristics that influence a designer's partitioning decision. These characteristics directly affect the methods by which an application is bisected to ensure robust software license protection.

7.1.1 Call Frequency

One of the most important decisions a designer must make when choosing which procedure to make private is how frequently that procedure is called by public regions of an application.

Any robust software licensing scheme must call private procedures fairly often to ensure that only an authorized computing system is used to execute the application. In the extreme, if a private procedure is only ever called once at the beginning of execution, then an adversary could simply capture the application state after that procedure call returns, and execute the application on any system he chooses. Recalling that Temporal Application Operation Equivalence is defined for some unit of time ω, *any* authentic application APP^{Auth} which calls a private procedure every σ time units is always at least "T-AOE for time σ" with respect to any corresponding counterfeit application APP^{Cf}. Since an adversary either fails or succeeds in emulating a private procedure on each call, it can crudely be said that the expected T-AOE time is $(\sigma \cdot P_{call})$, where P_{call} is the probability that one particular call is able to be emulated.

Unfortunately, the more frequent a private procedure is called, the slower an application will run. Although this speed is dependent on the particular secure architecture in use, as a general rule private code takes longer to execute than public code. (No matter what architecture is used, executing private encrypted code will always consume more resources, such as power or time.) While the designer must be aware of this slowdown, the speed difference is not always that considerable. For example, if an application's private code consumed 5% of the execution time of a program, and that code ran at 80% the speed of public code, then adjusting the ratio of private code execution time from 5% to 10% would result in a 1% overall performance degradation. Therefore under many circumstances it may be wise to increase the amount of private code in an application, which would likely decrease the chance of success of a counterfeiting adversary, while only suffering a slight performance degradation. For other high-performance applications, this may not make sense.

7.1.2 Input Data Transfer

Beyond frequency, the number of input arguments and values that are given to a private procedure affects the ability of an adversary to successfully emulate the procedure (cf. Section 6.2). Therefore a designer might decrease the chances of an adversary producing a counterfeit application by expanding the private procedure to encompass other procedures that require many inputs.

Unfortunately, it is also the case that the number of inputs which are given to a private procedure also affects the speed of execution. For example, in a secure coprocessor system, any transfer of data from public memory to private memory (including memory reads and writes) is likely to incur a large performance penalty. In the AEGIS processor, excessive arguments cannot be kept in processor registers and must be copied between public and private memory regions.

Not only do the number of input arguments affect the performance, but the number of input values can affect the hit rate of private caches. This, however, is unlikely to have a severe impact. Therefore it is often better for a designer to choose private procedures that accept few arguments with many differing values, rather than procedures which accept many arguments with few values.

7.1.3 Call Tree Location

Another design decision a partitioned application creator should address is the choice of whether to privatize "leaf" procedures which do not call any other procedures, or whether to privatize "trunk" procedures which make calls back to public procedures. From a performance standpoint, trunk procedures are likely to slow down the application as a whole since they introduce more transitions between public and private execution. On the other hand, the chances of an adversary counterfeiting a trunk procedure may be lower than that of a leaf procedure. This is because there are more outputs of the procedure that must be emulated.

7.2 Weighted DFG/CFG Bisection

Although there are any number of techniques that an application designer can use to determine how to partition his application [53], this section will briefly discuss one simple empirical method named *Weighted Data-Flow-Graph/Control-Flow-Graph Bisection.* While the bisection method presented is not at all complicated, it does present a feel for how the problem of partitioning can be solved, and what features matter most.

This application partitioning method addresses most of the important factors that affect security and performance, separating the problem into two phases of analysis. First, a completed application must be analyzed by its designer to construct a Control Flow Graph (CFG) that is weighted by call frequency, and a Data Flow Graph (DFG) which is weighted by the number of values that pass between procedures. Second, the designer uses a parameterized metric to analyze both graphs and pick a single bisection line that separates the application into public and private procedures.

During this discussion, it is assumed that there are no "trade-secret" procedures which *must* be privatized. This allows for a partitioning methodology which is free to choose any procedure it wants to be public or private. With this in mind, the scheme described is meant to convert a fully public application into a partitioned application that can only be executed by a single computing system (that is, ensuring software license protection).

7.2.1 Graph Construction

A Weighted Control Flow Graph (W-CFG) is simply a call graph that identifies which procedures call who, and how often. As can be seen in Figure 7-1, each node of this graph represents a procedure and each solid directed edge represents a procedural call, weighted by the number of times the call was made. Dashed edges are procedural returns.

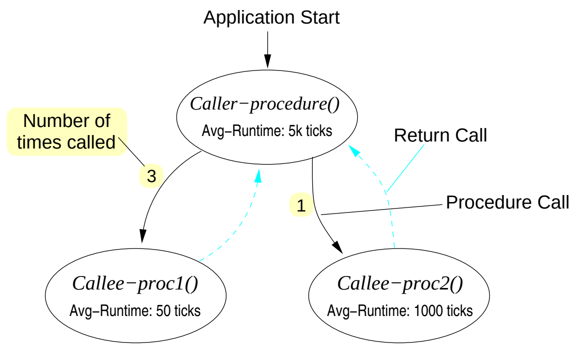


Figure 7-1: Weighted Control Flow Graph (W-CFG).

This graph can either be constructed statically, through procedural analysis done at compile time, or dynamically, by running an application on some workload that is characteristic of real-world use. Although a CFG is easy to construct statically [72], determining the weights for edges is highly dependent on the workload. Therefore it may be best for a designer to simply execute an application on a large set of workloads, recording the number of procedural calls using a functional simulator or a binary instrumentation tool [62]. It is certainly possible to estimate these weights, however, this work does not address such algorithms.

Figure 7-2 shows a magnified portion of a W-CFG for the *Gzip* application found in the

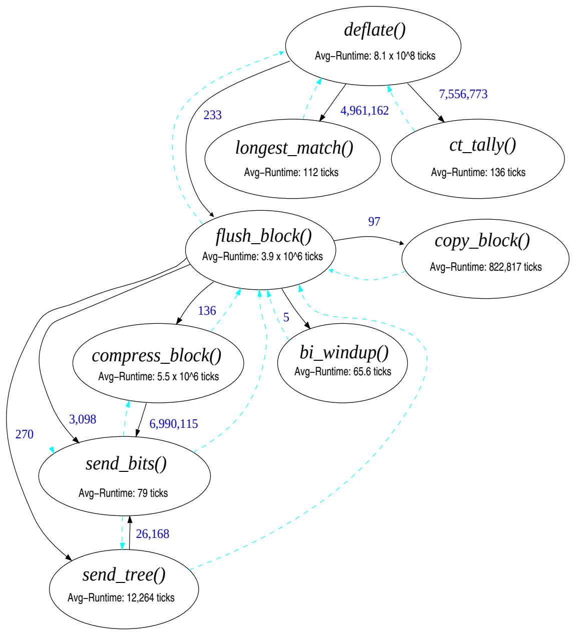


Figure 7-2: Magnified region of W-CFG for *Gzip*.

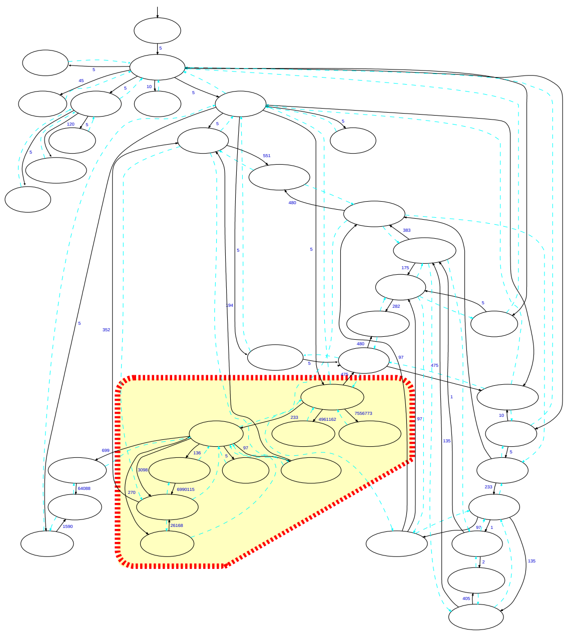


Figure 7-3: Entire W-CFG for *Gzip*.

SPEC CPU2000 benchmark suite. Figure 7-3 shows the W-CFG of the entire application. To generate these graphs, we simulated execution of *Gzip* on the five SPEC CPU2000 workloads provided for this benchmark using a chunk size of 3MB. (The workloads are `ref.random`, `ref.graphic`, `ref.program`, `ref.source`, and `ref.log`.) The procedure name is listed within each node as well as the average number of cycles it takes for the procedure to complete ("Avg-Runtime").

A Weighted Data Flow Graph (W-DFG) displays the flow of information between procedures, and was informally introduced in Section 6.3.1. Shown in Figure 7-4, each node again represents a procedure, while each directed edge signifies that one procedure has passed data to another. The weighting of each edge measures the number of unique Ad-

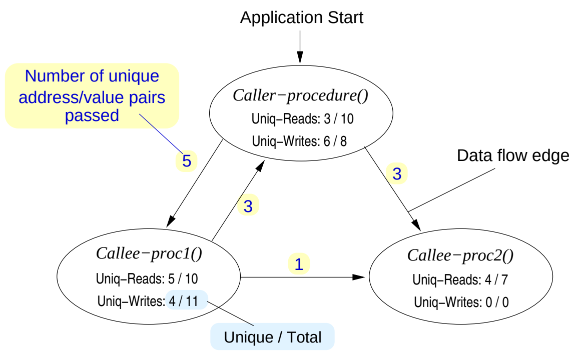


Figure 7-4: Weighted Data Flow Graph (W-DFG).

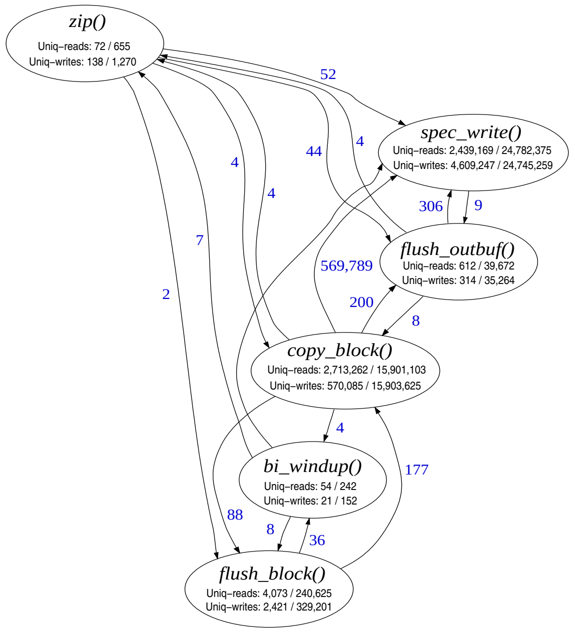

zip()
Uniq–reads: 72 / 655
Uniq–writes: 138 / 1,270
spec_write()
Uniq–reads: 2,439,169 / 24,782,375
Uniq–writes: 4,609,247 / 24,745,259
flush_outbuf()
Uniq–reads: 612 / 39,672
Uniq–writes: 314 / 35,264
copy_block()
Uniq–reads: 2,713,262 / 15,901,103
Uniq–writes: 570,085 / 15,903,625
bi_windup()
Uniq–reads: 54 / 242
Uniq–writes: 21 / 152
flush_block()
Uniq–reads: 4,073 / 240,625
Uniq–writes: 2,421 / 329,201
52
4
44
4
4
7
2
306
9
569,789
200
8
4
177
88
8
36

Figure 7-5: Magnified region of W-DFG for *Gzip*.

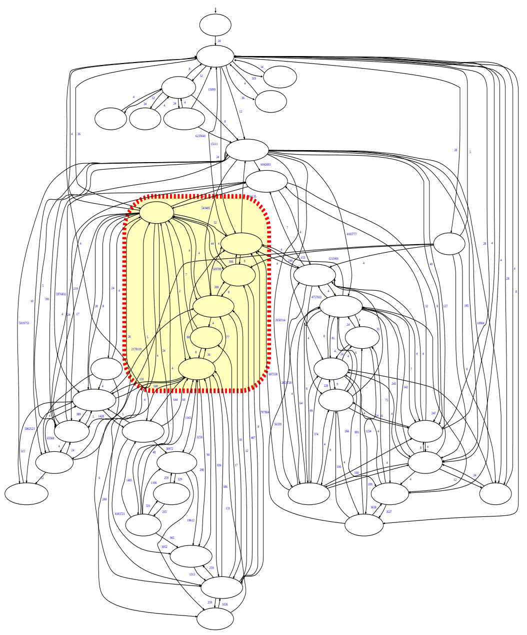


Figure 7-6: Entire W-DFG for *Gzip*.

dress/Value pairs that have been passed. For the AEGIS system, we define data passage between procedures as any AV pair which is written by one procedure and read by another. Unlike a CFG, a DFG cannot be perfectly constructed during compilation because of the inter-procedural aliasing problem [16, 72]. Further, determining the weight for each edge is again highly dependent on the input workload. Just as before, it may be possible to statically estimate these values, however, we shall only discuss examples where the W-DFG is constructed through execution of an application on a large number of workloads.

Figure 7-5 shows a magnified portion of a W-DFG for the *Gzip* application, while Figure 7-6 shows the entire W-DFG. This graph was generated in the same way as the W-CFG graph found in Figure 7-3. Listed within each node is the name of the procedure, as well as the ratio of the number of unique input AV pairs observed versus the total number of inputs ("Uniq-Inputs"). Similarly, the number of unique output AV pairs observed versus the total number of outputs is also given ("Uniq-Outputs"). Notice that it is possible for the number of unique outputs listed within each node to be smaller than the sum of all the weights of outgoing edges. This is because each edge weight represents the number of unique values that are passed to a particular function, thus a the same output AV pair can be counted twice in the edge weights if it is passed to two different procedures.

7.2.2 Bisection Metric

After the W-CFG and W-DFG graphs have been constructed, a designer can analyze these graphs to make an informed decision on how to partition an application in a way that promises robust software license protection. To do this he chooses some metric that will find a *cut* through the edges of both graphs which separates the public and private procedures. As an example, a primitive metric is discussed here using a simple flow based algorithm to find a satisfactory cut. Note that other bisection metrics are possible and can likely provide even better software licensing protection. For example, this method's use of a W-CFG used the *total* number of times a procedure is called but does not necessarily quantify the call *frequency* during the execution of an application.

To begin, the W-DFG and W-CFG are combined to form a single directed graph G. This is possible since both graphs share the same set of vertex procedure nodes V. To create the edges of the combined graph G, the union of the edges of W-DFG and W-CFG is taken, possibly combining edges that connect the same two vertices in the same direction (preventing a multi-graph). The weights of all edges are then determined by some function $W(\cdot)$ of the weights found in the W-DFG and the weights found in the W-CFG. For example, the weight of an edge between two vertices v_1 and v_2 can be determined by a simple linear function

$$W(E_{v_1 \to v_2}) \quad = \quad \alpha \cdot W_{DFG}(E_{v_1 \to v_2}) - (1 - \alpha) \cdot W_{CFG}(E_{v_1 \to v_2})$$

This function returns a high weight for edges that connect procedures which share many unique values, but penalizes procedures that are called many times. Therefore highly weighted edges can likely make good transition points from public to private execution.

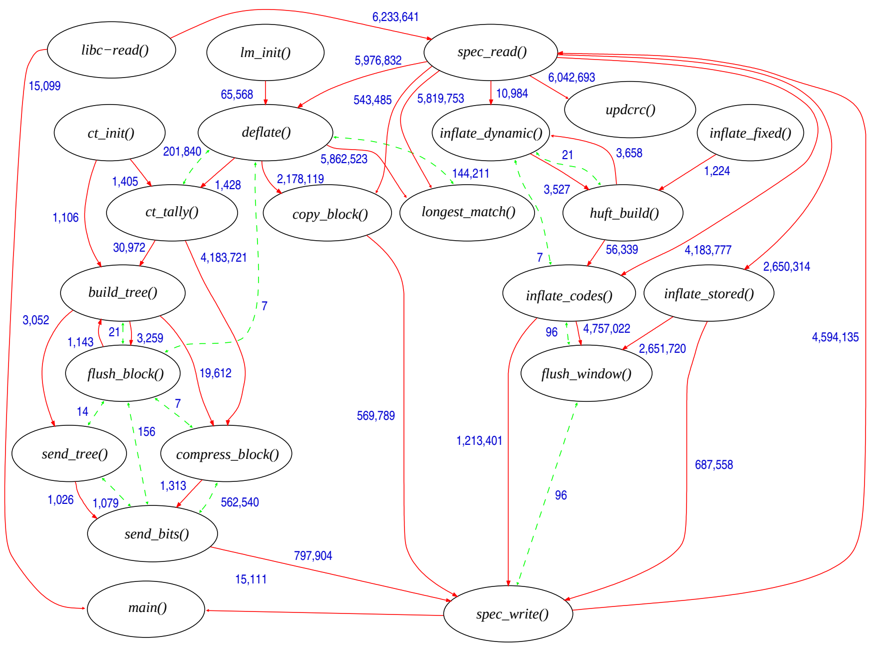


Figure 7-7: Combination of W-DFG and W-CFG for 22 procedures in *Gzip*.

The variable α is a constant parameter in the range $[0, 1]$ which signifies the importance of the number of procedure calls versus number of data values shared between procedures.

Given this W-DFG/W-CFG graph G, a designer selects every valid combination of vertices as a "source" and a "sink," and determines some cut between the two with a value greater than a security parameter ϑ. (The value of a cut is the additive combination of the weights of all edges that the cut crosses.) Once a cut is determined, the procedures that are on the "source" side of the cut can be made private. Since the "MAX-CUT" problem is NP-complete, and enumerating every possible source and sink combination can take exponential time, it makes sense to use some approximation algorithm, or to lower the value of ϑ until the partitioning decision can be made in a reasonable amount of time.

To give an example, 22 nodes were selected from the W-DFG graph of *Gzip* which have interconnecting edge weights of at least $1,000$. (Again, this data was generated by running *Gzip* on its five reference workloads using a 3 MB chunk size.) Figure 7-7 shows this subsection of the *Gzip* W-DFG graph. Superimposed are the edges of the W-CFG graph for *Gzip* that correspond to these 22 nodes. Procedure calls and returns are combined into one bidirectional edge.

Figure 7-8 shows the combined graph G with edges weights determined by $W(\cdot)$ above using the parameter $\alpha = 0.01$. If we assume that the "I/O" procedures `spec_read()`, `spec_write()`, and `libc-read()` must be contained within the public region of code, then one reasonable bisection is given by the shaded region surround by dashed lines. This cut separates four procedures to be made into one large private procedure and satisfies a security

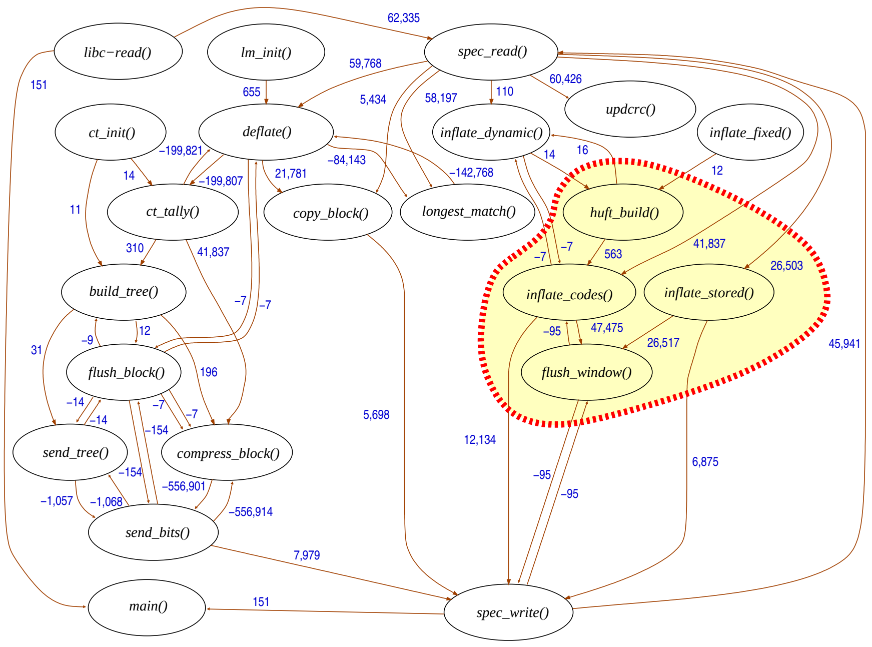


Figure 7-8: *Gzip* combined graph G with public/private bisection.

parameter of $\vartheta = 19,000$. This makes a good cut because it avoids as many procedure calls as possible while still retaining a large number of unique values entering and leaving the private region. Avoiding procedure calls that transition between private and public regions boosts performance, and increasing the number of unique values produced by the private region makes it harder for an adversary to perform a Memoization Attack.

Note, however, that this particular example has been simplified for ease of viewing, and is based only on procedure nodes which have high interconnecting weights in the W-DFG, and does not include all possible procedure call interactions. One would expect the W-CFG edges to play a greater role in determining the cut point in an analysis that examines the entire application.

Generally, given a large value of ϑ, the Weighted DFG/CFG Bisection method will most likely find one large portion of code that will have a high likelihood of resisting an emulation attack. Given a small value of ϑ, many sets of procedures may be acceptably secure partitions. In this case a designer can choose the set of procedures that consume the least amount of execution time according the average running time listed within procedure nodes. This will select a partition that performs well while still meeting the security requirement, ϑ.

Chapter 8

Conclusions

In conclusion, we present a brief summary of this work, review a number of important research topics that are not covered in this thesis, and part with some final thoughts on the problem of securely partitioning applications.

8.1 Summary

This work has presented a careful look at the inherent security problems caused by partitioning applications in a manner that ensures intellectual property privacy and software license protection. Specifically, two important questions have been investigates. First, how to ensure the secrecy of a small region within an application so that the functionality it serves within the application cannot be easily duplicated. Second, how to prevent the execution of a partitioned application by unauthorized parties.

These questions are examined under the assumption that an adversary is only able to monitor the input/output relationships of these small regions (private procedures) when performing an attack. This is a reasonable expectation if there is no extra-computational information known about the application under attack, such as human intuition. Further, the concept of Application Operation Equivalence is introduced as a way to disambiguate the ability of an adversary to reproduce the functionality of a private procedure and the ability of an adversary to reproduce the functionality of an entire application.

Given this adversarial model, it is shown that the optimum attack an adversary can perform is a Memoization Attack. This attack simply tabulates all input/output relationships that are observed during some prior execution of the application and replays specific relationships whenever necessary. Surprisingly, an implementation of this seemingly simply attack succeeds in duplicating the functionality of an application under a number of different circumstances. However, this attack can consume a large amount of resources and requires a number of implementation optimizations to be feasible at all.

Since it is possible for an adversary to successfully duplicate an application's functionality, it is important for an application designer to analyze newly created software in an attempt to detect possible vulnerabilities. A number of tests are proposed that identify private procedures which can be easily attacked. These tests can fairly accurately indicate when there might be problems, yet are efficient enough to be run on large applications.

Finally, the point is raised that security alone cannot guide an application designer's partitioning decision. Further consideration must be given to concepts such as procedure call frequency and execution speed to ensure a robust and practical software license protection. An example partitioning scheme is given which bisects a secure application such that the resulting partitioned application is resistant to Memoization Attacks as well as unauthorized execution.

8.2 Future Work

This has only been an initial step in the investigation of the security hazards inherent in partitioned applications. Specifically, the problems of ensuring the privacy of hidden procedures and protecting a partitioned application as a whole from unauthorized execution. However, a number of simplifying assumptions have been made throughout this work, including a focus on the AEGIS architecture, a restriction on what knowledge an adversary has at his disposal (inputs and outputs), a restriction on the type of attack (Monitor & Swap Attack), and a treatment of only one private procedure at a time.

Here we discuss further work that must be studied if a practical, efficient tool is to be made which can identify privacy vulnerabilities and emulation susceptibility in real-world applications, allowing for the automatic partitioning of software.

8.2.1 Private Libraries

This work has only considered adversaries who attempt to emulate a private region of code within a *single* partitioned application. From this, it is assumed that every hidden region of code is unique to only one application. It may be possible to mount more intelligent attacks if it was known that two applications shared the same private procedures, but used them in different ways. An example of this is private application libraries.

To address this naively, one can group all of the applications that use a private library into one composite-application, and discuss the Temporal Application Operation Equivalence of that entire combination of applications. However, in the end an adversary may only desire T-AOE for one of those applications, therefore a better analysis must take into consideration any possible information that can be gained from these other applications which an adversary is not trying to attack.

8.2.2 Versioning

Another issue that affects real-world applications is the existence of multiple versions of the same application code. Multiple versions of a private procedure can also exist if a bug-fix or a feature change is required. It is unclear, however, if the existence of two *nearly* identical private procedures can benefit an adversary. It may be possible for an adversary to identify the changes in the newer version of a private procedure, and use that knowledge to reduce the complexity of emulating the old version of the private procedure (such as by ignoring new feature inputs and only creating a counterfeit application which is T-AOE to the old version of the application).

8.2.3 Probabilistic Application Operation Equivalence

Another broad area which this work does not examine is the possibility that an adversary is satisfied with only *partial* application operation equivalence. That is, the final output of an authentic or counterfeit partitioned application can be erroneous. A common example of this would be a video player which does not need to decode every video frame correctly to still be useful. Another example is Failure-Oblivious Computing [57].

Expanding the definition of Application Operation Equivalence to account for acceptable application errors is necessary to properly secure a number of practical applications. However, the admission of application errors may dramatically increase the powers of an adversary and the likelihood of a successful emulation attack.

8.2.4 Input Selection for Active Adversaries

The adversarial model presented in this work allows for either passive or active adversaries to attack a private region of code (cf. Section 3.4). However, the Temporal Memoization Attack proposed only focuses on a passive adversary who simply observes the execution of an existing partitioned application. An active adversary is able to choose the sets of inputs that are fed to a private procedure. This can possibly increase the chances of success during the emulation of a private procedure if the sets of inputs are chosen well.

Unfortunately, choosing sets of inputs that maximize an adversary's chance of success when emulating a private procedure is a complex problem. A human-guided input selection method would likely perform well, however it would be an interesting study to examine a collection of more general algorithmic processes.

8.2.5 Panoptic Adversaries

Finally, one of the larger simplifications used in this work was the assumption that an adversary only analyzes the local interactions between public and private procedures (the input/output relationship pairs). Panoptic adversaries have been introduced as adversaries that are able to analyze both the local interactions between private and public procedures, as well as the global interactions between all public procedures (cf. Section 3.3.1). These global relationships can have meaningful implications which may help reveal the purpose of private procedures.

Furthermore, from the start we have ignored the possibility of "prior human knowledge" which can guide an adversary during an attack. Any such knowledge is highly specific to the application under attack, however it may be possible to identify classes of partitioned applications or classes of private regions that have a common theme that makes the private procedures easier to emulate. Such a taxonomy of applications and attacks might prove useful for commercial software development.

8.3 Final Comments

The central question of whether or not an adversary can duplicate an unseen procedure is problematic at best. Adding to this, the notion that the purpose of this duplication is to execute a larger application only complicates matters. The fundamental issue here can be summarized as model definition. The security of a system can only be *guaranteed* in terms of the model proposed.

General models can say very little about a practical situation. It is extremely difficult to embody "human knowledge" in a general model even though this is often the most important factor when attempting to recreate the functionality of a hidden procedure. A security question phrased with these generalized assumptions can easily lead to vacuous assurances. Information theoretic concepts and statistical learning theory can only say so much at this point, and frequently what they say is too abstract for real-world applications.

Specific models can only focus on one particular application and one particular attack. While it may be possible to make strong security promises for this model, it is of limited realistic value. New attacks are conceived every day, and the work required to develop a specific model might be useless for the next application to come along.

Given these two extremes, this work attempts to find a sensible middle ground, suggesting a model that is general enough to handle any application, while specific enough to make reasonable security statements. While a more specific definition of the capabilities of an adversary might result in more satisfying privacy assertions, such a model would be useless for a greater number of possible attacks. Similarly a more abstract or generalized model could have called upon stronger theorems, however the resulting work might be of little use to an application designer creating a piece of software.

The model settled upon in this work is a careful combination of practical attack scenarios, and tangible privacy guarantees. However, at this point it is even hard to verify if the combination chosen is near "correct." The ultimate goal is for this model to serve as a practically useful base for work that identifies the innate security concerns which are of great importance to partitioned applications.

Bibliography

[1] McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[2] S. T. Kent. *Protecting Externally Supplied Software in Small Computers*. PhD thesis, Massachusetts Institute of Technology, 1980.

[3] Enrique I. Oviedo. Control Flow, Data Flow, and Program Complexity. In *Proceedings of COMPSAC*, pages 145–152, 1980.

[4] Warren A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. *SIGPLAN Not.*, 16(3):63–74, 1981.

[5] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, September 1981.

[6] C. Morgan. How Can We Stop Software Piracy. *BYTE*, 6(5):6–10, May 1981.

[7] Tim Maude and Derwent Maude. Hardware protection against software piracy. *Communications of the ACM*, 27(9):950–959, 1984.

[8] Kuo-Chung Tai. A program complexity metric based on data flow information in control graphs. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, pages 239–248. IEEE Press, 1984.

[9] James R. Gosler. Software protection: Myth or reality? In *CRYPTO'85 Advances in Cryptography*, pages 140–157, 1986. Lecture Notes in Computer Science No. 218.

[10] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 182–194, 1987.

[11] Amir Herzberg and Shlomit S. Pinter. Public protection of software. *ACM Transactions on Computer Systems*, 5(4):371–393, 1987.

[12] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[13] Song C. Choi and Walt Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, 7(1):66–71, 1990.

[14] Steve R. White and Liam Comerford. Abyss: An architecture for software protection. *IEEE Transactions on Software Engineering*, 16(6):619–629, 1990.

[15] Eric J. Byrne. Software reverse engineering: a case study. *Software Practice and Experience*, 21(12):1349–1364, 1991.

[16] William Landi. *Interprocedural Aliasing in the Presence of Pointers*. PhD thesis, Rutgers University, 1992.

[17] John C. Munson and Taghi M. Khoshgoftaar. Measurement of data structure complexity. *Journal of System Software*, 20(3):217–225, 1993.

[18] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[19] J. D. Tygar and Bennet Yee. Dyad: A system for using physically secure coprocessors. In *IP Workshop Proceedings*, 1994.

[20] Bennet S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.

[21] David Aucsmith. Tamper Resistant Software: An Implementation. In *Proceeding of the 1st Information Hiding Workshop*, 1996.

[22] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[23] Ross Anderson and Markus Kuhn. Low cost attacks on tamper resistant devices. In *IWSP: International Workshop on Security Protocols, LNCS*, 1997.

[24] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.
`http://www.cs.auckland.ac.nz/∼collberg/Research/`
`Publications/CollbergThomborsonLow97a/index.html`.

[25] Markus Kuhn. The TrustNo1 Cryptoprocessor Concept. Technical Report, Purdue University, April 1997, 1997.

[26] U. G. Wilhelm. Increasing privacy in mobile communication systems using cryptographically protected objects. In *Verläßliche IT-Systeme 1997*, pages 319–334, November 1997.

[27] Hongji Yang, Paul Luker, and William C. Chu. Measuring abstractness for reverse engineering in a re-engineering tool. In *1997 International Conference on Software Maintenance (ICSM '97)*, October 1997.

[28] Tomas Sander and Christian F. Tschudin. "on software protection via function hiding". *Lecture Notes in Computer Science*, 1525:111–123, 1998.

[29] Tomas Sander and Christian F. Tschudin. Towards mobile cryptography. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1998.

[30] Tanguy Gilmont, Jean-Didier Legat, and Jean-Jacques Quisquater. Enhancing security in the memory management unit. In *EUROMICRO*, pages 1449–, 1999.

[31] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. *Lecture Notes in Computer Science*, 1666:388–397, 1999.

[32] S. W. Smith and S. H. Weingart. Building a High-Performance, Programmable Secure Coprocessor. In *Computer Networks (Special Issue on Computer Network Security)*, volume 31, pages 831–860, April 1999.

[33] Johan Agat. Transforming out timing leaks. In 27^{th} *ACM Principles of Programming Languages*, January 2000.

[34] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, July 2000.

[35] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the* 9^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

[36] NIST Special Publication 800-22. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Information Technology Laboratory of the National Institute of Standards and Technology, May 2000.

[37] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *Advances in cryptology - CRYPTO '01, Lecture Notes in Computer Science*, 2139:1–18, 2001.

[38] Ramarathnam Venkatesan, Vijay V. Vazirani, and Saurabh Sinha. A graph theoretic approach to software watermarking. In *IHW '01: Proceedings of the 4th International Workshop on Information Hiding*, pages 157–168, 2001.

[39] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation: Tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, 2002.

[40] Markus Jakobsson and Michael K. Reiter. Discouraging software piracy using software aging. In *DRM '01: Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, pages 1–12, 2002.

[41] Rudi Lutz. Recovering high-level structure of software systems using a minimum description length principle. In *Artificial Intelligence and Cognitive Science, 13th Irish International Conference, AICS2002*, September 2002.

[42] George Marsaglia and Wai Wan Tsang. Some difficult-to-pass tests of randomness. *Journal of Statistical Software*, 7(3):1–8, 2002.

[43] Thomas S. Messerges, Ezzat A. Dabbish, and Robert H. Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Transactions on Computers*, 51(5):541–552, 2002.

[44] A. J. Klein Osowski and David J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, 1, 2002.

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

[46] Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A trusted open platform. *Computer*, 36(7):55–62, 2003.

[47] S. Kim, K. Umeno, and A. Hasegawa. On the NIST Statistical Test Suite for Randomness. In *IEICE Technical Report, Vol. 103, No. 449, pp. 21-27*, 2003.

[48] David Lie, John Mitchell, Chandramohan A. Thekkath, and Mark Horowitz. Specifying and verifying hardware for tamper-resistant software. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 166, 2003.

[49] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 178–192, 2003.

[50] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the* 17^{th} *Int'l Conference on Supercomputing (MIT-CSAIL-CSG-Memo-474 is an updated version)*, June 2003.

[51] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings of the* 36^{th} *Int'l Symposium on Microarchitecture*, pages 339–350, December 2003.

[52] Lan Gao Jun Yang and Youtao Zhang. Fast Secure Processor for Inhibiting Software Piracty and Tampering. In *36th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2003.

[53] Xiangyu Zhang and Rajiv Gupta. Hiding program slices for software security. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 325–336, 2003.

[54] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[55] Trusted Computing Group. TCG Specification Architecture Overview Revision 1.2. `http://www.trustedcomputinggroup.com/home`, 2004.

[56] Ben Lynn, Manoj Prabhakaran, and Amit Sahai. Positive results and techniques for obfuscation. In *EUROCRYPT*, pages 20–39, 2004.

[57] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr. William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, December 2004.

[58] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: an infrastructure for efficiently protecting information leakage on the address bus. *SIGPLAN Not.*, 39(11):72–84, 2004.

[59] OpenRISC 1000 Project. http://www.opencores.org/projects.cgi/web/or1k, 2005.

[60] Microsoft Corporation. Technical Overview of Windows Rights Management Services. Microsoft white paper, April 2005.

[61] Ruby B. Lee, Peter C. S. Kwan, John P. McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for protecting critical secrets in microprocessors. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 2–13, 2005.

[62] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, 2005.

[63] Microcosm. Dinkey dongle. http://www.microcosm.co.uk/, 2005.

[64] Microsoft. Next-Generation Secure Computing Base.
`http://www.microsoft.com/resources/ngscb/defaul.mspx`, 2005.

[65] Netquartz. http://www.netquartz.com, 2005.

[66] Ishan Sachdev. Development of a Programming Model for the AEGIS Secure Processor. Master's thesis, Massachusetts Institute of Technology, May 2005.

[67] G. Edward Suh. *AEGIS: A Single-Chip Secure Processor.* PhD thesis, Massachusetts Institute of Technology, 2005.

[68] 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* 32^{nd} *Annual International Symposium on Computer Architecture (MIT-CSAIL-CSG-Memo-483 is an updated version available at http://csg.csail.mit.edu/pubs/memos/Memo-483/Memo-483.pdf)*, June 2005.

[69] Hoeteck Wee. On obfuscating point functions. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 523–532, May 2005.

[70] Youtao Zhang, Jun Yang, Yongjing Lin, and Lan Gao. Architectural Support for Protecting user Privacy on Trusted Processors. *SIGARCH Comput. Archit. News*, 33(1):118–123, 2005.

[71] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory.* John Wiley and Sons, 1991.

[72] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., 1997.

[73] Vladimir N. Vapnik. *Statistical Learning Theory.* John Wiley and Sons, 1998.

[74] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory, Second Edition.* Springer, 1999.