Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2006-080 December 8, 2006

Memoization Attacks and Copy Protection in Partitioned Applications

Charles W. O’Donnell, G. Edward Suh,, Marten vn Dijk, and Srinivas Devadas

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Memoization Attacks and Copy Protection in Partitioned Applications

Charles W. O'Donnell, G. Edward Suh, Marten van Dijk, and Srinivas Devadas [*]
Computer Science and Artificial Intelligence Laboratory (CSAIL)
Massachusetts Institute of Technology
Cambridge, MA 02139
{cwo,suh,marten,devadas}@mit.edu

ABSTRACT

Application source code protection is a major concern for software architects today. Secure platforms have been proposed that protect the secrecy of application algorithms and enforce copy protection assurances. Unfortunately, these capabilities incur a sizeable performance overhead. Partitioning an application into secure and insecure regions can help diminish these overheads but invalidates guarantees of code secrecy and copy protection.

This work examines one of the problems of partitioning an application into public and private regions, the ability of an adversary to recreate those private regions. To our knowledge, it is the first to analyze this problem when considering application operation as a whole. Looking at the fundamentals of the issue, we analyze one of the simplest attacks possible, a "Memoization Attack." We implement an efficient Memoization Attack and discuss necessary techniques that limit storage and computation consumption. Experimentation reveals that certain classes of real-world applications are vulnerable to Memoization Attacks. To protect against such an attack, we propose a set of indicator tests that enable an application designer to identify susceptible application code regions.

1. INTRODUCTION

Proprietary software architects have always been concerned with software piracy as well as the protection of trade-secret algorithms. However, the ease with which the Internet has allowed Intellectual Property (IP) to be stolen has made system and application security a first order concern for both hardware and software designers. This is not only a problem of individuals knowingly using an unlicensed application. But often the ability to "crack" an application can lead directly to the creation of viruses and Trojan horses that can affect any number of otherwise honest users. Therefore, application designers frequently go to great lengths to ensure the secrecy of application code.

In this work we investigate one of the most basic methods for an adversary to determine the functionality of hidden application code, a "*Memoization Attack*." This attack is meant to succeed against some of the security systems recently proposed that protect applications by partitioning code into public and private regions of execution [30, 46]. We have implemented a Memoization Attack, run it against a number of applications, and developed methods of identifying when an arbitrary application might be vulnerable to such an attack.

Surely, the most secure way to enable copy protection and maintain code secrecy is to run an entire application on some kind of *Trusted Computing Base* (TCB), only communicating the final outputs to the user when necessary. However, in practice this would place too much of a burden on the TCB. For example, secure processors can be thought of as a TCB that protects sensitive regions of code and data through encryption. A unique secret can be embedded within a processor and used to decrypt and execute an application when loaded. Unfortunately, whole-application encryption can inhibit the use of shared libraries, complicate upgrades and patches, and most importantly, force cryptographic resources to be used during the execution of an entire application. Whether the cryptographic logic exists in hardware or software, there will be sizeable performance and power penalties to pay. Since typically only a small portion of an application is considered sensitive IP, it does not make sense to suffer losses from protecting the entire thing. Therefore, a partitioned application remedies these issues by only requiring a TCB to execute the sensitive IP, and leaves all other code to be executed by conventional means.

Application partitioning is a well studied concept that aims to protect software IP by simply separating all or most of the sensitive areas of code, and privately executing them on a TCB. Systems have been developed that can do this using secure processors [30, 46], secure co-processors [55], dongles [38], or remote servers [14]. The key requirement of these systems is only a general purpose computing resource that can secretly execute code in the face of any kinds of attack [2, 27, 37].

While it seems self-evident that running an entire piece of software on a TCB guarantees its IP privacy and licensing assurances, it is not clear if these guarantees hold true for individual partitions of a partitioned application. If only a sub-region of an application is run on a TCB, it may be possible for an adversary to reconstruct that unknown application code. Such a reconstruction would allow the adversary to create his own application that duplicates the functionality of the original, invalidating the application designer's IP. Therefore, it is of the utmost importance to analyze the different ways by which an adversary might recover the hidden application code found in partitioned applications.

In Section 2, we put forth a simple adversarial model. Section 3 defines what a Memoization Attack is, and shows that it is the "best" possible attack an adversary can mount

[*] This work is generously supported by ITRI, Taiwan.

given our model. Section 4 describes one practical and efficient implementation of this attack, and Section 5 describes when the attack can be effective. Using these insights, Section 6 proposes heuristic metrics that can be used to identify whether a partitioned region of application code is susceptible to an Memoization Attack. Section 7 discusses other work in this area and Section 8 concludes.

2. ATTACK MODEL

In this work, we restrict our focus to one of the simplest types of adversary imaginable. As will be described in Section 2.4, our adversary can only *observe* the execution of a partitioned application and then attempt to reconstruct the hidden regions that are run on a TCB using that observation. More sophisticated adversaries are easy to envision, however, we feel it prudent to explore a very basic model to its fullest extent. Further, the adversarial powers we describe here can be considered necessary for a number of more complex types of adversaries.

2.1 TCB and Partitioned Application Model

For the sake of clarity we will only focus on one type of TCB model so that we can describe more concretely what actions an adversary can and cannot take, and what constitutes a partitioned applications. To this end, we look at physically secure processors and co-processors [30, 46, 55] since these represent some of the most secure methods that exist for TCB code execution. Specifically, we choose the AEGIS secure architecture [46] because of its fairly straightforward protocol for the execution of partitioned applications. The remainder of this paper and all of our experiments assume this model for a TCB.

In the AEGIS secure architecture a partitioned application is merely a combination of *private* encrypted regions and *public* unencrypted regions of code that switch back and forth during execution using two distinct processor modes. Application memory is also separated into encrypted and unencrypted regions, conceptually forming private and public divisions of data and code. The encrypted portions of code can only run in a secure mode that decrypts instructions, executes them, and protects the secrecy and integrity of any private data these instructions operate on. While executing in this mode an adversary can only observe accesses to public data, and cannot observe or modify private data or program execution. Unencrypted portions of a partitioned application run in an insecure mode, with no protection of the data the instructions operate on.

For simplicity, we assume that *procedures* and the data structures they "*own*" are the fundamental units of public and private division. We also assume that procedures do not maintain state across calls within encrypted regions of memory. Disallowing a procedure to maintain encrypted state between calls allows for a more clean analysis and is fairly realistic for a large number of procedures within applications

2.2 What an Adversary Can Observe

Figure 1 depicts a fragment of a partitioned application while it is run on an AEGIS secure architecture. Progressing downward is an execution trace of an application as it switches from a public region of code to a private region and back. To reduce clutter, we only show reads and writes to main memory and do not show any other machine operations (such as add, etc.).

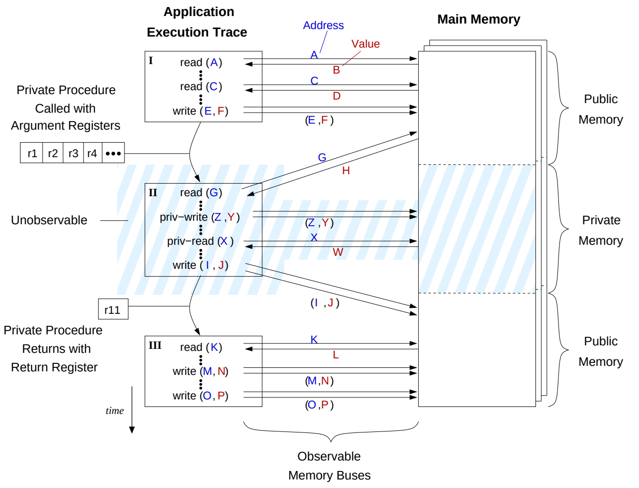


Figure 1: Visible and hidden procedure inputs and outputs in a partitioned application.

Beginning in box **I** a public region of code executes and performs some arbitrary procedure. Note that, this being a public procedure, all accesses to memory can only touch regions of memory that are also public. Since this is unencrypted code executing on a conventional processor, an adversary can inspect everything involved with the procedure. The procedure itself can be read to determine its control flow, the processor state can be examined cycle-by-cycle, and all memory requests and responses can be sniffed.

At the end of box **I** the procedure calls a private region of code (box **II**) and transfers control to the TCB to execute that private procedure. This call requires procedure arguments to be passed to the TCB, as shown by the registers "`r1`," etc. in Figure 1 (as defined by the application binary interface and including the stack and frame pointers). Similarly, once the procedure completes, a return value is also passed back from the TCB to the conventional processor. Since the private procedure was encrypted, an adversary cannot inspect the code directly to determine its control flow, nor can it examine the processor state cycle-by-cycle since its a TCB. Further, this TCB model *hides* any accesses to its private memory stack that the private procedure makes.

Therefore, the only information an adversary can observe relating to the private code is the arguments passed into the procedure, the return value passed back from the procedure, and any accesses to *public* memory that the private procedure makes (since public memory requests cannot be hidden by the TCB and the values within public memory are unencrypted). All three of these can be described as a collection of Address/Value (AV) pairs, where the "*Address*" indicates a memory address or argument register identifier, and the "*Value*" is the actual data being accessed. Once the private procedure returns to execution to public code (box **III**), the adversary can again observe everything.

2.3 Adversary Goals

Principally, IP secrecy and copy protection depends on preventing an adversary from discovering the contents of a partitioned application's private code. However, it is critical that we notice that an adversary does not need to *exactly* determine the contents of a private region of code, *but must only reproduce a private procedure's effect on the system sufficiently well as to allow the entire partitioned application to continue to function as designed*. Therefore, an adversary's most simple goal is to replace "*authentic*" private procedures with indistinguishable "*counterfeit*" procedures that can reproduce the adversary's desired "*functionality*" and "*utility*" of the partitioned application as a whole.

Ultimately, the only functionality that matters to an adversary is the set of application outputs that result from some set of inputs he is interested in. If the set of inputs are time-dependent, then the adversary may further only be interested in reproducing functionality for a limited amount of time. To this extent, an adversary need not *understand* each private procedure, but must only duplicate its external effects. For example, assume a fragment of an application performing the power function $f(x, p) = x^p$ is made secret. If an adversary only ever *cares* about executions when $p = 3$ then his only interest is in duplicating code that performs $f(x, 3) = x^3$.

Consequently, this limited sense of duplication of functionality is exactly what we should be concerned with when analyzing possible attacks on a partitioned application. This can be formally defined as *Temporal Application Operation Equivalence* (T-AOE).

Definition 1. **T-AOE**$(APP', APP'', \langle\boldsymbol{\Lambda}\rangle, t_s, \omega)$
Assume two applications APP' and APP'' begin execution at time 0 and finish execution at the same time H. During each unit of time between 0 and H, both applications are given the same, new vector of inputs $\boldsymbol{\Lambda}_t$ chosen from some set of many input vectors, the total available input set $\langle\boldsymbol{\Lambda}\rangle$. These applications are T-AOE at some time t_s for the length of time ω if, during the period $[t_s, t_s + \omega]$, the responses or results of both applications $\boldsymbol{\Psi}'_t$ and $\boldsymbol{\Psi}''_t$, are exactly equivalent (assuming $0 \le t_s \le H$ and $(t_s + \omega) \le H$).

Given this definition, the adversary we are concerned with aspires to create a counterfeit private region of code for a specific partitioned application that maximizes T-AOE time ω (ideally, $\omega \to (H - t_s)$). This ω can be thought of as the adversary's "time-till-failure."

2.4 Adversarial Powers

Unfortunately, any realistic adversary we try to model involves a human who has some *innate prior knowledge* about the application under attack that can make the process of recreating a hidden region of code trivial. For all we know the adversary may even be the author of the original source code for a private procedures.

Given the inability to formally capture such knowledge, we will simply treat a private procedure as a mathematical function with inputs and outputs the adversary is capable of observing. Our adversary has no understanding of the purpose of a private procedure and can only obtain knowledge of the code's functionality by observing the procedure arguments, the public memory accesses, and the procedure output of an authentic application run on a TCB. Note, although it seems probable for a real-world attack, our adversary does not analyze the available public code at all to infer any "meaning" to the application. This would again prove quite troublesome to model.

To formally specify these powers, we say that for each call to a private procedure, an adversary is aware of an *input set* $\boldsymbol{\lambda}$ of memory reads and the procedure's arguments, and an *output set* $\boldsymbol{\psi}$ of memory writes and the procedure's return value. These can be thought of as a vector of data values indexed by addresses or register numbers. Combining these vectors forms a single *input/output relationship pair* $(\boldsymbol{\lambda}, \boldsymbol{\psi})$. An adversary observing multiple calls to a private procedure can collect a multiple number of these pairs.

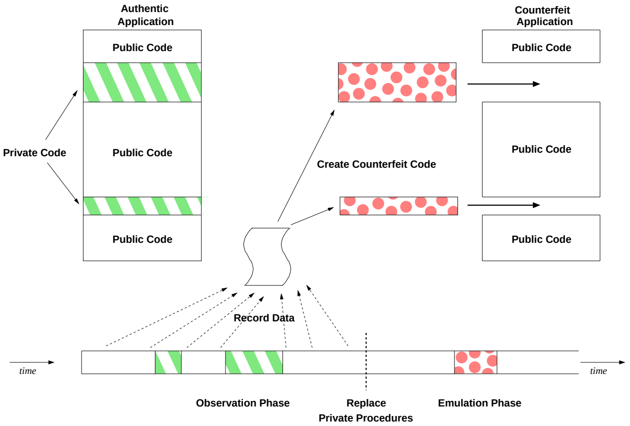


Figure 2: Basic technique a Memoization Attack.

3. MEMOIZATION ATTACKS

A "*Memoization Attack*" is the name we give to an attack that uses an authentic private procedure's input/output relationship pairs to create an alternate counterfeit version. Figure 2 shows one way a Memoization Attack can be performed. The adversary begins by running an authentic application using a TCB for some amount of time. During this time all input/output relationship pairs $(\boldsymbol{\lambda}, \boldsymbol{\psi})$ of the private procedures are observed and stored into a single "*Interaction Table*." At some point the adversary stops executing the authentic application on the TCB and constructs replacement private procedures using the interaction table that was captured. He can then continue to execute the application using these counterfeit private procedures. Whenever a counterfeit procedure is called, the set of inputs $\boldsymbol{\lambda}$ are read, and the interaction table is searched for a match. If a match is found the counterfeit procedure returns the corresponding output set $\boldsymbol{\psi}$, emulating the procedure, and continues execution of the application. Otherwise the application fails and terminates. The application continues running as long as calls to the counterfeit procedure are completed correctly, agreeing with our previous definition of failure under T-AOE.

To determine just how powerful this attack can be, let us first assume there exists an adversary with infinite memory and computational power, but who must also abide by the restrictions on adversarial powers discussed in Section 2. Note, while this adversary may have infinite general purpose computational power, we assume that he cannot decrypt private procedures and is restricted to the use of a real TCB to run authentic applications. Therefore what an adversary can observe from the execution of an authentic application remains the same as in a Memoization Attack since this is essentially defined by our model (although with infinite memory this adversary can save everything). Our question is then: can this omnipotent adversary mount a different type of attack that can outperform a Memoization Attack (that is, have a longer ω value for T-AOE).

Now let us assume that this adversary observes L calls to an authentic private procedure, somehow creates and inserts his own counterfeit procedure, and continues running the application. No matter how the counterfeit procedure is constructed, when the counterfeit procedure is called during emulation only one of two things can happen. If the exact set of inputs $\boldsymbol{\lambda}$ had been seen during the observation phase, then the adversary can simply return the set of corresponding outputs $\boldsymbol{\psi}$ that it had saved. However, if the set of inputs to the counterfeit procedure contain any new elements, then the adversary must rely on some other knowledge to decide what to output. Unfortunately, the only knowledge the adversary has is the set of input/output relationship pairs already seen. In the absence of any other knowledge, there's no reason to believe there's any correlation between prior input/output relationship pairs and the current input. Such a correlation requires a "prior" or an abstract learning model. Therefore this adversary's best option is simply to *uniformly guess* the values within the output set.

If the maximum number of outputs the procedure returns is σ, and the number of possible different outputs is κ, then the probability of the adversary guessing every output correctly is $\left(\frac{1}{\kappa}\right)^{\sigma}$. Assuming each set of inputs is uniformly selected from the set of all possible inputs, $\boldsymbol{\Gamma}$, the probability of an adversary correctly emulating a new call to a counterfeit procedure is

$$P_{call} = \left(\frac{L}{|\boldsymbol{\Gamma}|}\right) + \left(1 - \frac{L}{|\boldsymbol{\Gamma}|}\right)\left(\frac{1}{\kappa}\right)^{\sigma}.$$

Assuming an extremely large set $\boldsymbol{\Gamma}$, the probability of a guess being correct is practically zero, which makes this attack look quite the same as a Memoization Attack. Further, the probability of an adversary successfully emulating ω calls is simply a sequence of Bernoulli trials, so $P_{\omega} = (P_{call})^{\omega}$.

Therefore, even an adversary with access to unlimited memory and unlimited computational power cannot do better than simply memoizing every input/output relationship pair he observes. Given no innate knowledge of a private procedure, nor any idea of its output distribution, an attack as simple as a Memoization Attack is also the best attack possible.

Note, we do not consider adversaries with knowledge of a non-uniform output distribution of a private procedure. These adversaries may be able to increase their probability of success when guessing, but algorithmic effort to increase this probability likely exhibits diminishing returns. That is, it is our intuition that the computation time required to approximate the output distribution grows exponentially as a function of this approximation's accuracy.

4. IMPLEMENTING MEMOIZATION

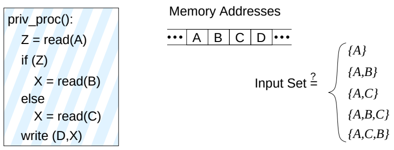
To investigate the feasibility of a Memoization Attack, we implemented a tool that is capable of observing the execution of a partitioned application, constructing an interaction table, replacing all private procedures with counterfeit procedures, and re-running the partitioned application on alternate inputs. Although building such a tool may sound easy, a naive implementation would require more computational and storage resources than are available on a standard computer workstation. Therefore, we present here some of the tactics used to streamline the implementation so that it can be executed efficiently.

4.1 System Setup

To create a tool that performs a Memoization Attack, we created a special functional simulator for our chosen TCB (an AEGIS processor). Binary applications are run on the simulator, using some input data set, while attack-specific tasks are performed in the background whenever a transition is encountered from a public to a private procedure. In an attack, the simulator is first started in an observation mode, which saves an interaction table to disk as the application is run. The simulator is then restarted, using a different input data set, and uses the saved interaction table to emulate all private procedures. An assembly rewriting tool was constructed to automate the separation of public and private procedures in compiled applications.

4.2 Creating an Interaction Table

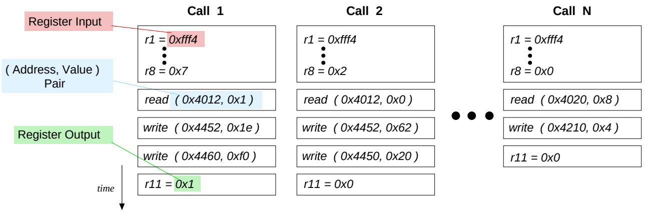
At first glance, creating an interaction table sounds quite simple: observe an application execute and create a flat mapping of each private procedure's inputs and corresponding outputs. However, emulating a private procedure with such a flat lookup does not work on real systems. The problem is: at the moment when a private procedure is first called an adversary cannot know all of the input values to that private procedure. This is because the procedure itself determines what memory addresses are to be read as inputs *during* execution. This is what we call "*Input Self-Determination*," and is demonstrated in Figure 3. As we can see in the figure, an adversary cannot know whether address B is actually part of the input set until we know the value at address A.



```
priv_proc():
    Z = read(A)
    if (Z)
        X = read(B)
    else
        X = read(C)
    write (D,X)
```

Figure 3: When procedure is called, full set of inputs are unknown due to Input Self-Determination.

Therefore, our interaction table must contain more information than just the input/output relationship pairs; the table must keep information about the *temporal ordering* of those pairs as they occurred during the execution of the authentic application. One way to visualize such a table is shown in Figure 4, where each "*column*" represents a call to the procedure which holds an ordered list of Address/Value pairs.

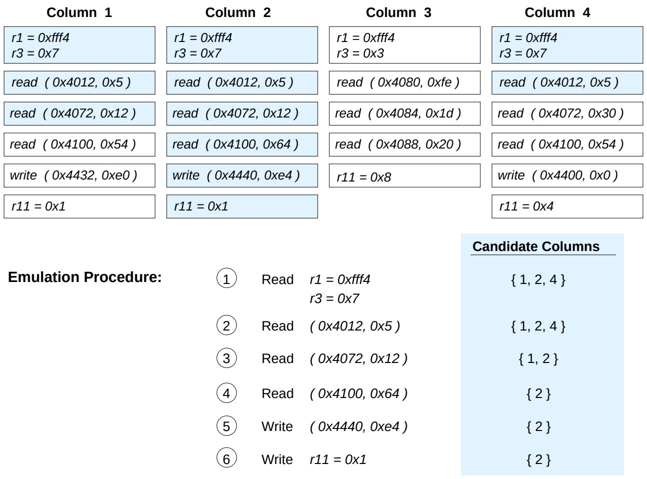


Call 1	Call 2	Call N
r1 = 0xfff4 ⋮ *r8 = 0x7*	*r1 = 0xfff4* ⋮ *r8 = 0x2*	*r1 = 0xfff4* ⋮ *r8 = 0x0*
read (0x4012, 0x1)	*read (0x4012, 0x0)*	*read (0x4020, 0x8)*
write (0x4452, 0x1e)	*write (0x4452, 0x62)*	*write (0x4210, 0x4)*
write (0x4460, 0xf0)	*write (0x4450, 0x20)*	*r11 = 0x0*
r11 = 0x1	*r11 = 0x0*	

Figure 4: Basic private procedure interaction table.

4.3 Emulation using an Interaction Table

Once an adversary has created an interaction table of the type discussed in Section 4.2, he can quite easily emulate



Column 1	Column 2	Column 3	Column 4

r1 = 0xfff4
r3 = 0x7
	r1 = 0xfff4
	r3 = 0x7
		r1 = 0xfff4
		r3 = 0x3
			r1 = 0xfff4
			r3 = 0x7

read (0x4012, 0x5)	read (0x4012, 0x5)	read (0x4080, 0xfe)	read (0x4012, 0x5)
read (0x4072, 0x12)	read (0x4072, 0x12)	read (0x4084, 0x1d)	read (0x4072, 0x30)
read (0x4100, 0x54)	read (0x4100, 0x64)	read (0x4088, 0x20)	read (0x4100, 0x54)
write (0x4432, 0xe0)	write (0x4440, 0xe4)	r11 = 0x8	write (0x4400, 0x0)
r11 = 0x1	r11 = 0x1		r11 = 0x4

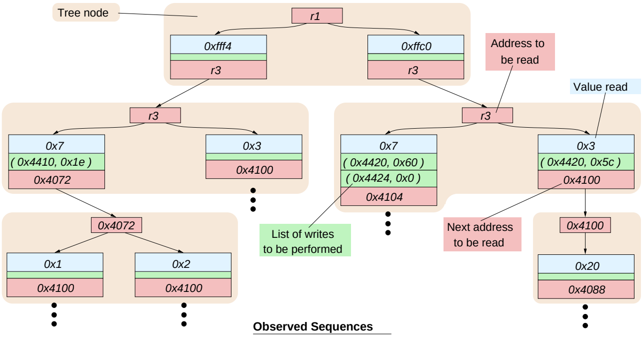
Emulation Procedure:

			Candidate Columns

1	Read	r1 = 0xfff4
		r3 = 0x7
			{ 1, 2, 4 }

2	Read	(0x4012, 0x5)	{ 1, 2, 4 }
3	Read	(0x4072, 0x12)	{ 1, 2 }
4	Read	(0x4100, 0x64)	{ 2 }
5	Write	(0x4440, 0xe4)	{ 2 }
6	Write	r11 = 0x1	{ 2 }

Figure 5: Emulation steps using a basic interaction table.



r1 = 0xfff4
r3 = 0x7
write (0x4410, 0x1e)
read (0x4072, 0x1)
read (0x4100, . . .)
. . .
	r1 = 0xffc0
	r3 = 0x7
	write (0x4420, 0x60)
	write (0x4424, 0x0)
	read (0x4104, . . .)
	. . .
		r1 = 0xfff4
		r3 = 0x7
		write (0x4410, 0x1e)
		read (0x4072, 0x2)
		read (0x4100, . . .)
		. . .
			r1 = 0xfff4
			r3 = 0x3
			read (0x4100, . . .)
			. . .
				r1 = 0xffc0
				r3 = 0x3
				write (0x4420, 0x5c)
				read (0x4100, 0x20)
				read (0x4088, . . .)
				. . .

Figure 6: An interaction tree holding the same information as in Figure 5, but in a compressed form.

any private procedure that is run on inputs he has already seen. We show how this emulation can be done in Figure 5. When a private procedure is called, the input arguments (registers **r1**, etc.) are matched against the previously seen arguments found at the beginning of each column (independent procedure call) in the table. The set of columns with matching arguments then constitutes a set of candidate previously observed procedure calls that the current procedure call might be an exact copy of. Notice, the next row of all the candidate columns is the same and dictates whether a memory read or write happened in the previously observed calls. If the next row is a write, the write is performed and the following row is inspected (which will still be the same for all columns). However, if the next row is a read, there exists a new input value and the set of candidate rows can possibly be reduced. This continues until a row contains the procedure's return value, in which case the emulation succeeds, or the set of candidate columns is reduced to zero, in which case the emulation fails.

4.4 Compressing an Interaction Table

The method for creating an interaction table described in Section 4.2 is sound and will work on procedures that have very few inputs or are called only a few times. However, because of the way it maintains order information, this table's size will grow unmanageably when dealing with procedures that have numerous inputs (many memory reads) or procedures that are called often with many different values for their inputs.

To solve this issue, we instead imagine the structure keeping track of the ordering of inputs and outputs as a tree. Instead of each column representing a unique call to the procedure, the root of the tree represents the beginning of any call to the private procedure and each branch leaving the root is one possible execution path. Such a tree can reduce the amount of *redundant* data found in our interaction table (as might have been noticed in Figure 5).

An example of what this tree might look like is shown in Figure 6. Notice that since only memory *reads* can change an execution path, each "tree node" contains the memory address of the next read that should be performed in that one execution path, as well as any writes that must be made before that next read is input. A private procedure can be emulated using this tree in much the same way as an interaction table is used as described in Section 4.3.

Although this tree drastically reduces the amount of data we must save to perform a Memoization Attack, it still contains some redundancies. The actual data structure used in the implementation of our attack tool is significantly more complicated, and is basically a full, possibly cyclic graph that intelligently keeps track of unique paths from a single start node to many possible end nodes.

The motivation for this is based on the following observation: when a tree keeps track of inputs and outputs to a private procedure, loops within a single call can create an extremely deep and repetitive tree, even when the same memory addresses and values are being read over and over. Further, often multiple calls to a private procedure that differ in their initial arguments can later exhibit identical input and output traces for long periods of time. For example, a procedure that takes two input arguments, computes many values using the first argument, but only uses the second argument at the very end of the procedure. When using a tree data structure this would create two separate, but *nearly* identical branches from the root.

Unfortunately, due to space constraints, we cannot describe exactly how this graph data structure is constructed and used for emulation, but only that it is based on a unique-numbering method that pinpoints divergences and convergences of execution traces. We have found that by using such a graph data structure, surprisingly less space will be consumed than with a tree data structure.

5. EFFECTIVENESS OF MEMOIZATION

After running our implementation of a Memoization Attack on a number of applications in the SPEC CPU2000 [22] suite, we found that two particular types of partitioned applications are susceptible to this attack. By this we mean that an adversary has a good chance of successfully emulating the private procedures of a partitioned application, given an arbitrary or naive partitioning of that application into public and private regions. The two classes of applications can be identified by the types of inputs they require to accomplish their task, applications with *Partially Repeated Input Sets* and applications with *Composite Input Sets*.

5.1 Applications with Partially Repeated Input Sets

The first class of partitioned applications we found to be susceptible to a Memoization Attack are those that have private procedures that are called when the exact same inputs are given to a single execution of an application over and over (such as a repeated common function like "save"), or those that have private procedures that are only ever called when the application reads identical inputs on every execution (such as fixed runtime flags or a common data set).

To illustrate this class, we examined the "*Parser*" application found in the SPEC CPU2000 suite, and assumed a partitioning scheme that only treats individual procedures as private or public. *Parser* executes in two stages, first it processes a fixed dictionary file, and second it analyzes sentences for their grammatic structure using that dictionary. At any time it also accepts special directives from a user that can perform the same operation over and over. Both traits (two stage execution and special directives) are indicative of an application with partially repeated input sets.

To test whether a Memoization Attack would succeed, we designated the `special_command()` procedure in `main.c` to be private. We then observed the application while sending the `!echo` directive (that sets whether to display output to the screen) which uses the `special_command()` procedure. *Parser* was then run on new input data and we were able to emulate the call the `!echo` without any problems.

Next, we made the `is_equal()` procedure in `read-dict.c` private and observed the application when run on the standard dictionary file and the input file "`smred.in`" taken from MinneSPEC [42]. This procedure is only called while *Parser* reads the dictionary file. In our attack, we were able to correctly emulate this procedure when executing the entire *Parser* application on much larger input files `mdred.in` and `lgred.in`. Both attacks proved successful. Further, Table 1 shows that the storage requirements for the interaction table (or actually graph, cf. Section 4.4) are not large at all.

Metric	*Parser* `special_command()`	*Parser* `is_equal()`
Total number of nodes in graph	283	5
Size on disk (in Bytes)	26,972	3,042,968
Maximum number of inputs per call	743	5

Table 1: Size of memoized private procedures.

5.2 Applications with Composite Input Sets

The second class of partitioned applications we found susceptible to Memoization Attacks are those that contain private deeply "inner" procedures (such as libraries) that are only ever fed a finite number of unique inputs (due to the control flow of the calling procedures), no matter what external inputs are given to the application as a whole. In this case a Memoization Attack might succeed by simply observing an authentic application run on *any* large set of inputs, hoping to "cover" or "saturate" the set of inputs to the inner procedure. Because these "saturating" procedures are often not immediately apparent (unlike, perhaps, those mentioned in Section 5.1) this class of applications presents a significant problem for a software architecture who would like to prevent Memoization Attacks.

To test whether a Memoization Attack would succeed on this class of applications we attempted to emulate a few procedures from the *Gzip* and *Parser* applications in the SPEC CPU2000 suite, assuming a partitioning scheme that only treats individual procedures as private or public. Table 2 summarizes the results.

In our attack of *Gzip*, we attempt to emulate a number of procedures using the input file `ref.log` after observing the execution of *Gzip* on just the `ref.random` input file, both the `ref.random` and `ref.graphic` input files, and so on. Even though there is virtually no overlap between these inputs, we found that the `bi_reverse()` procedure can be emulated almost entirely correctly. Of the $1,797$ calls made to `bi_reverse()` while processing `ref.log`, $1,741$ of the calls contained the *exact* same procedure inputs as had been observed when running *Gzip* on the first four input files.

Similarly, our attack on *Parser* attempted to emulate a number of procedures using the the `mdred.in` and `smred.in` input files after observing the execution of the application using the `lgred.in` input file. Although none of the procedures could be fully emulated after memoizing input/output relationships pairs from `lgred.in`, it is clear that there are still many duplicated procedure calls between the two unrelated input files. It might even be possible for an adversary to fully emulate the `contains_one()` procedure if he simply observes a large enough set of application inputs from an input file.

From this experimentation we see that a Memoization Attack may be able to succeed even when application inputs seen during emulation are completely unrelated to application inputs recorded during observations.

6. IDENTIFYING APPLICATIONS THAT ARE VULNERABLE TO ATTACK

In Section 5 we have shown that a Memoization Attack can succeed on certain classes of applications. This may be useful information for an attacker, however, we would rather help software architects *avoid* such attacks through their choice of what procedures to make private and what procedures to make public.

Ideally, we would like to have some *test* that tells us whether a particular private procedure can be easily emulated via a Memoization Attack. The simplest test could be to just run our a Memoization Attack on that procedure. However, to run this attack on all procedures in an application would be computationally infeasible. Instead, information theoretic analyses could be applied, but these might also prove ineffective for practical applications because of their assumptions on entropy, complexity, input space, and "learnability" may be too general. [13, 48, 49].

Thus, we propose the use of two heuristics, or "*indicators of insecurity*," that *speculate* upon the likelihood that a private procedure can be emulated in a partitioned application. Importantly, these indicators focus on the interaction of the procedure with the application, rather than the procedure itself. While these tests are not absolute, a procedure that passes them can be given a high confidence that it is immune to a Memoization Attack. Such methods of identifying negative results are used often in problems that do not have a clear positive indicator, for example, tests determining the

Gzip procedure (Lines of assembly)	Percentage of correct procedure calls while emulating `ref.log` after observing input set(s) `ref.*`			
	{`random`}	{`random,graphic`}	{`random, graphic,program`}	{`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 input set `lgred.in`	
	emulating `mdred.in`	emulating `smred.in`
`contains_one` (123)	33% (1136/3485)	0% (0/71)

Table 2: Success of a Memoization Attack on applications with composite input sets.

"randomness" of a random number generator [26, 33, 35].

6.1 Indicator 1: Input Saturation

Our first test, *Input Saturation* tracks whether a private procedure is only ever fed a finite number of distinct inputs (AV pairs) by the rest of its application. A simple way to detect this is to run an application using successively more inputs, and observe whether the number of distinct inputs fed to a procedure is linearly related to the number of inputs fed to the application, or if the number of distinct inputs fed to that procedure *saturates* at some level. Procedures that are input saturating are likely easy to emulate through a Memoization Attack (assuming a correlation between the number of unique input AV pairs and the total number of ordered sets of input AV pairs).

Many techniques exist that can estimate the number of unique input AV pairs given to a procedure, however, we simply created a tool that counts and efficiently stores this number while an application is run on a specific input set. Given a large enough input set, this method quickly separates input saturating procedures from those that are not.

As a case example we used this tool to identify input saturating procedures in the SPEC CPU2000 application *Gzip*. Figure 7 plots the number of unique AV pairs that are fed to the *Gzip* procedure `ct_tally()` during *Gzip*'s execution on five large, orthogonal input sets. For normalization purposes, the x-axis represents the number of calls made to the procedure instead of time. We call this a "*cumulative input density*" plot, and use it as a helpful visualization of when a procedure might be input saturating.

In Figure 7, we see that the rate of increase in the number of unique AV pairs decreases as more input sets are applied. In fact, the input set `ref.log` did not cause *any* new AV pairs to be fed to the procedure, implying that an adversary might be able to emulate `ct_tally()` on `ref.log` given the observation of the prior four input sets. (Note, in reality this fact would not guarantee a 100% successful attack since our metric does not account for input order, but the metric does correlated with a high level of success.)

To numerically quantify the information in the cumulative input density plot we can use two specific metrics. First, the "*average input delta*" ($Avg.I\Delta\%$) can tell us the percentage increase in the number of unique AV pairs input to a procedure from call to call. This gives an estimate of how many procedure calls are expected before a new input is seen, and correlates exactly to ω in the formulation of T-AOE in Section 2.3. Second, the "*saturation weight*" (SW) is a single number that gives an idea of the shape of the cumulative input density plot. If the function $w(c)$ represents the number of unique inputs for the c^{th} call out of N calls, then SW is the normalized integral

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

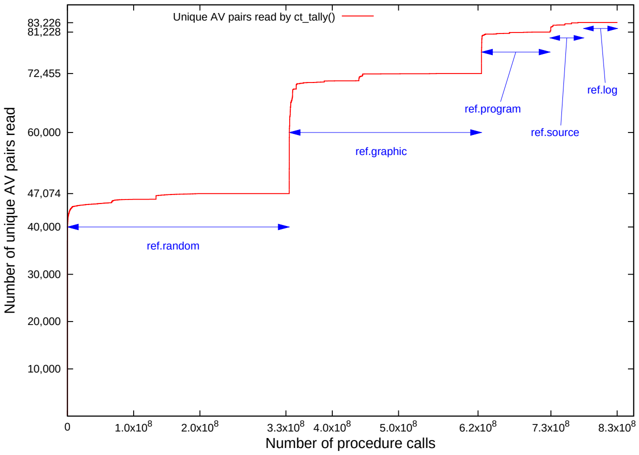


Figure 7: Cumulative input density plot of unique AV pairs for *Gzip*'s `ct_tally`, when run on a large input set.

Looking closer at our example, we've run *Gzip* on a smaller version of the same set of inputs and highlighted five of its procedures to demonstrate different levels of input saturation typical in applications. Figure 8 shows a cumulative input density plot for these procedures executing on the five inputs (normalizing the total number of calls and inputs between procedures to make comparison easier), and Table 3 gives their average input delta and saturation weight values.

By simply "eyeballing" the plots in Figure 8, we see that the `ct_tally()` and `bi_reverse()` procedures are probably input saturating, and would be susceptible to a Memoization Attack, while the `build_tree()` and `longest_match()` procedures are probably not. It is less clear if the `huft_build()` procedure is input saturating since it does not appear to plateau overall, but does plateau for each workload. We attempted a full Memoization Attack on `huft_build()` (Table 2) and found that we could not successfully emulate the procedure. This might lead us to say that only "clearly" saturating procedures should be labeled as vulnerable.

The SW values for these procedures also agree with our conjectures, giving values close to 1.0 for easily emulated procedures. However, the $Avg.I\Delta\%$ values do not show a correlation. This is important to note: $Avg.I\Delta\%$ only estimates ω for one specific procedure and has little meaning when used for comparison.

Finally, Table 3 also shows the average input delta and saturation weight values of the *Gzip* `ct_tally()` procedure using the larger version of the input set. Over the smaller input set, this larger input set causes a drastic decrease in SW from 0.87 to 0.77, lowering it to levels near `huft_build()`

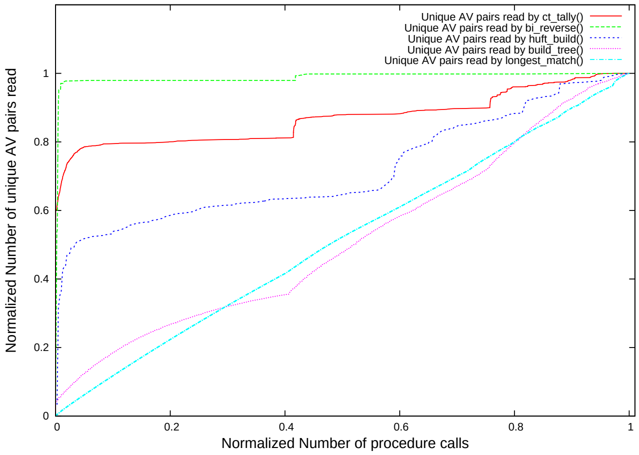


Figure 8: Cumulative input density plots from *Gzip*.

(0.72). However, `huft_build()` shows very little susceptibility to a Memoization Attack and `ct_tally()` shows very high susceptibility (as mentioned earlier, no new inputs are seen when the input set `ref.log` is applied). This implies that our original statement that only "clearly" saturating procedures are vulnerable should be rethought, and, more importantly, this stresses the need for a conservative interpretation of these heuristic metrics when making any security decisions.

6.2 Indicator 2: Output Weighting

Our second test, *Output Weighting*, tracks the values that a private procedure outputs and those values' usefulness to the application as a whole. In essence, this determines how important a private procedure is to the entire application. Output Weighting is possibly a better test than input saturation since it indicates how important a private procedure is to the entire application.

Since an adversary only cares about whole-application functionality, partitioning an application by making less important procedures private may lead to a more successful Memoization Attack. For example, assume during a memoization attack that an adversary cannot return the correct outputs for a private procedure call but continues running. If the previous values in memory still produce the correct behavior (because of range checks, etc.) then the adversary will still be content. This "low importance" of the outputs of the private procedure has allowed a Memoization Attack to succeed. Another good example arises when a private procedure's outputs are only ever used by a single, simple public procedure that is itself easily emulated. In this case, the inputs and outputs of the private procedure do not matter, and a Memoization Attack can succeed by simply emulating the simple public procedure wrapping the private procedure. The output weighting of a private procedure should be able to identify both cases as susceptible to a Memoization Attack.

Procedure	Tot. uniq. reads	Tot. uniq. writes	Public readers	$\Phi(\cdot)$ 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 4: Output weights for six *Gzip* procedures.

Tracing the entire flow of data throughout an application and deriving "usefulness" information is again a hard task. Therefore we suggest a simple test that estimates how much "usefulness" public procedures derive from the outputs of a private procedure. This test recognizes that a private procedure can only impact the outputs of the entire application if its own outputs are passed along and or used by other public procedures. This test is called the the "*output weight*" $\Phi(\cdot)$ of a procedure, and is defined by

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

Here $\boldsymbol{\eta}$ is a set of pairs (ι, κ), where ι is the number of unique outputs written by a private procedure and read by a public procedure, and κ is of the total number of unique outputs from that public procedure. For example, if five public procedures use the outputs of one private procedure as their input, then $|\boldsymbol{\eta}| = 5$. Here the fraction $\frac{\kappa_i}{\iota_i}$ indicates the impact of a private procedure's outputs on the outputs of public procedures. In other words, this indicates how the utility of a private procedure's outputs is "*amplified*" as the outputs are used by the rest of the application. A very large value of $\frac{\kappa_i}{\iota_i}$ implies that a private procedure's outputs are important.

As with input saturation, the output weight of a procedure can be estimated using many techniques. However, for simplicity we made a tool that efficiently tabulated the number of unique outputs that are transferred between procedures while an application is run on some input set. From these tabulations we can compute the output weight, as shown in Figure 9 where the number of unique outputs out of the `inflate_codes()` procedure in *Gzip* are used to determine an output weight of $\Phi(\cdot) = 390,657$.

Looking again at our *Gzip* example, Table 4 gives the computed output weight of six select procedures from an execution of *Gzip* on the small version of the same five input sets. We also show the number of unique reads and write a private procedure performs, and the number of pub-

Procedure	Total unique inputs seen after execution on the input set(s) `ref.*`					*Avg.* *I*Δ%	*SW*

	`{random}`	`{random,graphic}`	`{random,graphic,`
			`program}`
				`{random,graphic,`
				`program,source}`
					`{random,graphic,`
					`program,source,log}`		

`ct_tally` (large input)	47,074	72,455	81,228	83,226	83,226	9.7x10^{-9}	0.77
`ct_tally` (small input)	2,304	2,550	2,768	2,836	2,837	6.9x10^{-7}	0.87
`bi_reverse`	569	580	580	580	581	6.3x10^{-5}	0.99
`huft_build`	0	2,500	3,170	3,510	3,586	7.4x10^{-3}	0.72
`build_tree`	11,873	23,611	29,945	32,103	32,672	5.9x10^{-3}	0.51
`longest_match`	4.78 M	8.33 M	10.13 M	11.19 M	11.61 M	2.7x10^{-6}	0.51

Table 3: Rate of input saturation for five *Gzip* procedures.

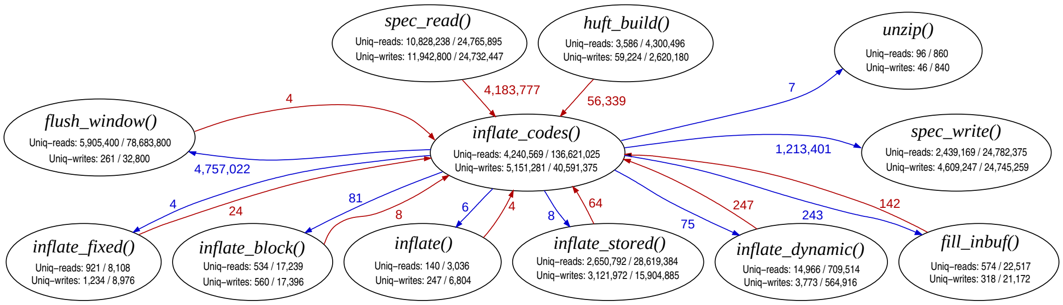

spec_read()
Uniq–reads: 10,828,238 / 24,765,895
Uniq–writes: 11,942,800 / 24,732,447
huft_build()
Uniq–reads: 3,586 / 4,300,496
Uniq–writes: 59,224 / 2,620,180
unzip()
Uniq–reads: 96 / 860
Uniq–writes: 46 / 840
flush_window()
Uniq–reads: 5,905,400 / 78,683,800
Uniq–writes: 261 / 32,800
inflate_codes()
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
4,183,777
56,339
4
4,757,022
7
1,213,401
4
24
81
8
6
4
8
64
75
247
243
142

Figure 9: The unique outputs of the `inflate_codes()` procedure in *Gzip* which can be used to compute its output weighting. of $\Phi(\cdot) = 390, 657$.

lic procedures that read what that private procedure writes.

We see from Table 4 that the procedures **`inflate_codes()`** and **`ct_tally()`** produce many unique outputs that are then fed to other procedures that in turn produce many unique outputs . If this trend continues, it is highly likely that the outputs of the application as a whole will depend significantly on the outputs of **`inflate_codes()`** and **`ct_tally()`**. Alternately, the outputs of 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 do not produce many more unique outputs. Therefore, it might be easy for an adversary to perform a Memoization function on these latter four procedures

6.3 Interpreting Indicators

The two indicators presented are not absolute and are not the only possible metrics of whether an application is susceptible to a Memoization Attack. In practice a software architect should apply as many tests as possible, including more complicated tests, before feeling confident that a private procedure is safe from attack. Further, the results from such tests should be weighted in tandem. As we see in our examples, the set of "safe" procedures that the input saturation test determined does not perfectly overlap with the set of "safe" procedures the output weighting test determined. Ultimately, no test can rule out the possibility of a Memoization Attack since this attack depends directly on the input set applied to the application. Therefore, the amount of testing performed is yet another design choice when deciding how to partition an application into private and public regions.

7. RELATED WORK

Only a few studies [6, 8, 9, 19, 31] have specifically examined software secrecy and modification of application code to prevent an adversary from determining its contents, sometimes suggesting techniques with which to decipher these contents. To counter such techniques *obfuscation* transforms have been proposed that make an application incomprehensible, but still functionally correct [10, 11]. Unfortunately, it has been proven that cryptographically secure obfuscation is generally impossible for a large family of functions [4] (although a few specific families have been shown to be obfuscatable [32, 51]).

A more popular way of concealing application instructions is through encryption. Homomorphic encryption schemes [44, 45] allow meaningful computations to be performed on encrypted data, but are not general enough for practical use. Instead, many have suggested using a small trusted computing base to decrypt ciphertext applications and to execute instructions confidentially [3, 25, 34, 40]. This idea of using specialized security hardware and secure coprocessors has seen many manifestations [16, 28, 54, 55, 58].

Recently, physically secure architectures have been proposed that reduce this trusted computing base to a single chip while also supporting application partitioning [30, 46]. These allow applications to be encrypted and executed on a processor without revealing any information to even the device owner. Even though these architectures encrypt application instructions, additional methods must still be employed to defend against side-channel attacks [1, 17, 18, 59].

Application encryption can be used for copy protection (by bind software execution to a specific key), but concepts of watermarking [11, 50], renewability [24], online-verification [12, 14], and hardware assisted authentication [15, 20, 29, 39] have also all been suggested as means to enforce basic software licensing. Unfortunately, many of these methods suffer from the same fundamental problem: they add *extra* code to the application. While it may be extremely difficult, a motivated adversary can almost always remove this extra code.

Architectural support for application partitioning is not a new concept [52, 55], however we believe that this paper is the first analysis of the problem of code secrecy when considering application operation as a whole. Program slicing has been proposed [57] as a means to prevent piracy, however it does not address the possibility that program secrecy may not be guaranteed in a partitioned application. Other compiler and language support for secure partitioning has been proposed [5, 56], but focuses on a different problem of application etiquette and information flow.

Finally, the indicators discussed are basically an analysis of code complexity. Many empirical software complexity metrics have been developed over the years [7, 21, 23, 36, 41, 43, 47]. Of these, one [53] does discuss the complexity of deconstructing an application, but does not focus on security.

8. CONCLUSIONS

Application partitioning has been suggested by a number of works as a means to allow an application to run efficiently on a TCB while preserving the TCB's security guarantees. We have investigated the problem of maintaining IP secrecy and copy protection in a partitioned application by looking at how to prevent an adversary from duplicating the *functionality* or *utility* of a private partition. Importantly, this analysis often depends more on the make-up of the entire application than it does on the make-up of a private partition. This is formalized by the adversarial goal of Temporal Application Operation Equivalence.

To begin to tackle this question we have analyzed one of the simplest forms of attack, a Memoization Attack. This attack simply tabulates input/output relationships seen during legitimate executions of a private partition, and replays what had been saved when it later emulates that private partition. Under certain assumptions this attack is the best an adversary can possibly perform.

We implemented a full Memoization Attack, and described some necessary techniques that allow this attack to run with reasonable storage and computation time restrictions. By running this attack on real-world applications, two classes of partitioned applications were found that are susceptible to Memoization Attacks.

To help software architects identify these classes of partitioned applications during development we proposed two efficient tests that can be used to identify an application's susceptibility to a Memoization Attack. These tests were implemented efficiently and run on an example application to demonstrate their usefulness.

This has only been an initial step in the investigation of the security hazards inherent in partitioned applications. There are also many complicating issues that can be explored on this topic, such as the ease with which private libraries can be attacked (since multiple input sets are used in a predictable way), or if multiple versions of an application can makes it any easier on an adversary.

Ultimately, the question of whether or not an adversary can duplicate an unseen private partition is problematic at best. The exact security of a system can only be guaranteed in terms of the model proposed. Further, such concepts like "human intuition" do not easily fit into models, even though this is often the most important factor when performing such attacks. Therefore, our approach in answering this question is to begin with a simple, practical, but universal attack model that can then be built upon by more complicated attack models that address specific domains of human-injected knowledge.

9. REFERENCES

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

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

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

[4] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *Advances in cryptology - CRYPTO '01, Lecture Notes in Computer Science*, 2139:1–18, 2001.

[5] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, Aug. 2004.

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

[7] 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.

[8] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

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

[10] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

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

[12] M. Corporation. Technical Overview of Windows Rights Management Services. Microsoft white paper, Apr. 2005.

[13] T. M. Cover and J. A. Thomas. *Elements of Information Theory.* John Wiley and Sons, 1991.

[14] O. Dvir, M. Herlihy, and N. N. Shavit. Virtual Leashing: Internet-Based Software Piracy Protection. In *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 283–292, 2005.

[15] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open platform. *Computer*, 36(7):55–62, 2003.

[16] T. Gilmont, J.-D. Legat, and J.-J. Quisquater. Enhancing security in the memory management unit. In *EUROMICRO*, pages 1449–, 1999.

[17] 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.

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

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

[20] T. C. Group. TCG Specification Architecture Overview Revision 1.2. http://www.trustedcomputinggroup.com/home, 2004.

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

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

[23] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, Sept. 1981.

[24] M. Jakobsson and M. 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.

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

[26] 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.

[27] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. *Lecture Notes in Computer Science*, 1666:388–397, 1999.

[28] M. Kuhn. The TrustNo1 Cryptoprocessor Concept. Technical Report, Purdue University, April 1997, 1997.

[29] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. 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.

[30] D. Lie. *Architectural Support for Copy and Tamper-Resistant Software.* PhD thesis, Stanford University, 2003.

[31] R. 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*, Sept. 2002.

[32] B. Lynn, M. Prabhakaran, and A. Sahai. Positive results and techniques for obfuscation. In *EUROCRYPT*, pages 20–39, 2004.

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

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

[35] 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.

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

[37] T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Examining

smart-card security under the threat of power analysis attacks. *IEEE Transactions on Computers*, 51(5):541–552, 2002.

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

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

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

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

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

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

[44] T. Sander and C. F. Tschudin. ”on software protection via function hiding”. *Lecture Notes in Computer Science*, 1525:111–123, 1998.

[45] T. Sander and C. F. Tschudin. Towards mobile cryptography. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1998.

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

[47] K.-C. 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.

[48] V. N. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, 1998.

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

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

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

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

[53] H. Yang, P. Luker, and W. C. Chu. Measuring abstractness for reverse engineering in a re-engineering tool. In *1997 International Conference on Software Maintenance (ICSM ’97)*, Oct. 1997.

[54] L. G. J. Yang and Y. Zhang. Fast Secure Processor for Inhibiting Software Piracty and Tampering. In *36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003.

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

[56] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3):283–328, 2002.

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

[58] Y. Zhang, J. Yang, Y. Lin, and L. Gao. Architectural Support for Protecting user Privacy on Trusted Processors. *SIGARCH Comput. Archit. News*, 33(1):118–123, 2005.

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