Hardware Assisted Control Flow Obfuscation for Embedded Processors

Xiaotong Zhuang Tao Zhang Hsien-Hsin S. Lee Santosh Pande

College of Computing
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA, 30332-0280
{xt2000, zhangtao, leehs, santosh}@cc.gatech.edu

ABSTRACT

With more applications being deployed on embedded platforms, software protection becomes increasingly important. This problem is crucial on embedded systems like financial transaction terminals, pay-TV access-control decoders, where adversaries may easily gain full physical accesses to the systems and critical algorithms must be protected from being cracked. However, as this paper points out that protecting software with either encryption or obfuscation cannot completely preclude the control flow information from being leaked. Encryption has been widely studied and employed as a traditional approach for software protection, however, the control flow information is not 100% hidden with solely encrypting the code. On the other hand, pure software-based obfuscation has been proved inefficient to protect software due to its lack of theoretical foundation and considerable performance overhead introduced by complicated transformations. Moreover, even though obfuscation can prevent static reverse engineering, attacker can still successfully bypass the obfuscation by monitoring the dynamic program execution.

To address all of these shortcomings, this paper presents a hardware assisted obfuscation technique that is capable of obfuscating the control flow information dynamically. Dynamic obfuscation changes memory access sequence on-the-fly and conceals recurrent instruction access sequences from being identified. Our scheme makes it provably difficult for the attacker to extract any useful information. Our results show that a high-level security protection is possible with only minor performance penalty. Finally, we show that our scheme can be implemented on embedded systems with very little hardware overhead.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Miscellaneous;
K6. [**Management of Computing and Information Systems**]: Security and Protection;

C3. [**Special-purpose and application-based Systems**]: Real-time and embedded systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'04, September 22–25, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-890-3/04/0009…$5.00.

General Terms: Design, Performance.

Keywords: Obfuscation, Control Flow Graph

1. INTRODUCTION

With the performance of embedded systems continuing to grow, more complex applications are expected to run on embedded platforms. State-of-the-art embedded processors such as Intel's XScale processor family or StarCore LLC's SC1400 DSP (or Motorola's SC140e) can run at very high frequency with a rich instruction set support and execute some large applications originally designed for desktop systems.

Due to the ever-growing deployment of embedded systems and their intrinsic vulnerability, software protection on embedded systems is emerging as a challenging problem to avoid adversaries to easily gain a full physical access to the system hardware, internal critical application algorithms, and sensitive data. Leaking the software may directly result in the loss of critical information and intellectual property (IP) coming from tens or hundreds of man-year's software development effort and investment. Given that embedded software development can be more time-consuming due to its special properties like meeting real-time constraints, specific memory capacity limitation, low-level interfaces, etc., the protection of the software IP is highly desirable.

It is common to achieve software copy protection via various encryption techniques. The security strength of encryption is provable, depending on the encryption scheme and the length of the key. Traditionally, encryption-based methods have performance concerns since the program code and data must be decrypted before they can be processed. If the decryption is based on software, the performance degradation may be too substantial to be tolerable to users. With the rapid advancement of IC fabrication technology, hardware-based encryption support is becoming widely available in modern security devices, such as smart cards, midsize secure devices [5] etc. With the help of dedicated hardware decryption engines, the decryption operation can be accelerated substantially to alleviate the performance degradation from using software-based methods. Due to the recent advances in security devices with hardware support for encryption/decryption, it is reasonable to predict that encryption will play a major role in the future.

Employing the encryption alone, however, is not a sufficient solution to software protection in most cases. Even though the plaintext of program code and data are completely

"unrecognizable" from the attacker after encryption, their absolute and relative locations are not altered at all. In other words, if we want to get the same data or code block again, the same address must be issued on the bus. As will be introduced in section 3, the control flow information will be left unchanged. A classical way to conceal the control flow information is through obfuscation. The goal of obfuscation is to transform the program from its original form to a functionally equivalent one that is more difficult to understand and reverse engineer. Conventionally, obfuscation involves software techniques like layout obfuscation, control obfuscation, data obfuscation etc [2].

However, obfuscation has several weaknesses. The most important one is that it lacks of theoretical foundation. There is no solid way to measure and prove the difficulty introduced after the transformation, i.e. the level of protection cannot be evaluated and guaranteed quantitatively after the obfuscation. On the other hand, it has been proved that a perfect obfuscation does not exist after all [3]. It is well acknowledged that an experienced programmer can foil all kinds of obfuscation schemes if enough time is given. Actually, the main task of obfuscation is to prevent an automated de-obfuscator. Moreover, obfuscation may incur large overheads in the code size due to dead code or irrelevant code [2] , if a sophisticated transformation is chosen. Also, the performance may be penalized severely due to carrying out extra obfuscating computations. Therefore, even traditional obfuscation techniques like "inlining and outlining transformation", "loop transformation", "control flow flattening" [2][3] can obfuscate the control flow to some extent, however they are purely software-based and thus suffer from the problems as mentioned above. Finally, the history has proven obfuscation is inefficient in protecting software. After being first introduced twenty years ago, it has been developed very slowly and is rarely used in reality, due to the aforementioned reasons.

Consequently the state-of-the-art hardware-based encryption can achieve low-overhead but cannot obfuscate at the control flow level, while the software-based obfuscators can obfuscate the control flow but are worse in performance. Besides, quantitatively there is no guarantee on how much protection can be provided with software obfuscation.

In this paper, we propose a hardware based approach to obfuscate the program control flow at runtime with very small overhead. Based on existing hardware encryption techniques, hardware assisted control flow obfuscation provides a much higher level of obfuscation than its software-based counterpart. Through the theoretical analysis, we show that the probability for an attacker to successfully identify recurring addresses, an important criteria in revealing the control flow information, is extremely low.

The rest of the paper is organized as follows: section 2 discusses the system model; section 3 depicts IP piracy issues due to unprotected control flow; section 4 introduces preliminaries; section 5 presents hardware assisted obfuscation; section 6 shows security analysis of our scheme; section 7 gives additional considerations; section 8 shows and analyzes experimental results; section 9 and section 10 gives related work and our conclusion.

2. SYSTEM MODEL

We assume an embedded processor, which supports hardware based encryption mechanism. The processor chip is physically secure, such that all data and code cannot be cracked once they are brought onto the processor chip. On the other hand, data and code are always encrypted whenever leaving the processor and need to be decrypted after they are fetched into the processor. Therefore, only the processor chip is trusted, whereas buses and external memory are subject to attack. Such a secure processor model was well studied in literature [6][7][8][9] and was implemented in commercial products such as Dallas Semiconductor's DS5002FP microcontroller [5] in which the processor chip is physically shielded and code and data are encrypted outside the chip. More recently, similar security models are proposed and designed at the architecture level, e.g. eXecute Only Memory (XOM) from Stanford [10]. With proper hardware implementations, techniques proposed in [11][12] show that encryption and decryption can be efficiently performed at an inexpensive hardware cost.

3. IP PIRACY DUE TO UNPROTECTED CONTROL FLOW

This section explains why encryption alone cannot prevent the exposure of program control flow and how this can cause IP piracy.

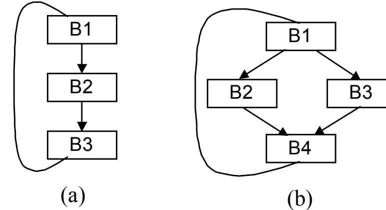

Figure 1. Identifying control flow structures.

First of all, encryption techniques can only encrypt the contents of the memory blocks, making the attacker unable to interpret them correctly. However, the location of the blocks are not changed. In other words, using the current memory model, if the block at address 100 is accessed the first time, all the subsequent references to the same data will have to fetch it from address 100. Even if the attacker does not know what is exactly stored in address 100, he or she can still confidently know that the same block is being used again and again when the same address is observed during a program's execution. Now we illustrate two examples with regards to how a loop and a change-of-flow (i.e. a branch) can be exposed. We assume all the blocks are encrypted but their locations in memory are never changed. As shown in Figure 1.a where three blocks [1] form a loop, then we will see addr(B1), addr(B2), addr(B3) repeat many times. On the other hand, branches can be identified as well. In Figure 1.b that shows a conditional branch resides inside a loop and both branches are taken during the execution, we will see that addr(B1), addr(B2), addr(B4) and addr(B1), addr(B3), addr(B4) appear on the bus. Thus, the attacker can reasonably believe there is a branch at block 1 and the two branch targets are block 2 and block 3, respectively. Although in this example, the detection of the branch relies on the behavior of a loop, even if there is no loop encompassing a branch, the branch can still be identified by executing the code multiple times experimenting with different input data. As such, the attacker can incrementally gather most of the control flow graph

[1] In this paper, we assume control flow graphs are constructed at block level, i.e. each node is a block, and edges indicate control flow transitions between blocks.

by monitoring the stream of addresses for a sufficient amount of time. Up to now, we have only illustrated that how a control flow graph could be reconstructed, at least partially, but not the actual contents of the code.

It is undeniable that the control flow information is the essential part of the algorithms implemented in software. Previous software obfuscation techniques mostly aim to prevent the control flow information from being obtained by the attacker. For example, obfuscation techniques such as "converting reducible to non-reducible flow graph", "inline and outline", "loop transformation" [2][3], etc., change the control flow directly. Unfortunately, encryption alone cannot prevent the leakage of the control flow, although this kind of IP piracy may not expose the software in its entirety, its damage can be severe under the following scenarios.

Imagine company A and company B are developing similar software. Company A succeeds first and releases the software in encrypted format so it can be run on a processor with hardware encryption/decryption. However, company B now has a complete access to the software and can run it repeated on the processor and experiment with it extensively, e.g. feed the program with different inputs, run that software together with their own software on the same machine or even with a manipulated OS. Moreover, company B is already an expert in developing similar software, therefore it only wants to understand a few critical parts or algorithms of the software. Therefore, they can definitely benefit from the control flow information as extracted in a similar manner as shown in the previous examples. Although this scenario is not as straightforward and prevalent as software piracy by the end users, it has been a major concern of software companies (like the recent Linux/SCO-Unix lawsuit). Besides this, one can steal critical data (like secret key) through side-channel attacks as conducted in many low-end systems including smart-card, ATM terminal devices.

Also as discussed in [15][21], control flow information can help to identify reuse code, leading to its complete exposure. This is done by matching the control flow graph against control flow graphs of the code that are already known to the attacker. As the reuse code takes an ever-increasing portion (more than half) in modern software given the fast time-to-market pressure and a large amount of legacy code that can be reused, leaking reuse code could severely endanger software protection. In addition, the values of some data might be indirectly exposed by the control flow as well. For example, if a branch compares variable x with 1 and decides which path to take, we will know whether x actually equals 1 by observing the direction of the branch. Although most likely, only partial information can be obtained through the control flow revelation, such information can be valuable in certain circumstances regardless.

However, software-based obfuscation can only avoid such IP piracy to a limited extent. Control obfuscation techniques like "inline and outline", "loop transformation" are well-known techniques. If the attackers knows the obfuscator has changed the code, he can somehow reverse that process to find the original or a close-to original form of the code. In [4], obfuscation is provably not a cure-all solution, the traditional software-based obfuscation approaches cannot even clarify the level of protection provided, i.e. they cannot quantitatively guarantee how much protection a scheme provides after the obfuscation is applied. Moreover, software-based obfuscations are conducted at compilation time, which means once the obfuscation is done, the code will not be changed any more at runtime. Therefore, the attacker can experiment with the same code repeatedly, making it more likely that some useful information will be eventually revealed. Finally, as mentioned earlier, software-based obfuscation typically has ill-effects like code growth, extra computation etc., which may deteriorate the quality of the original code.

It is noteworthy that caches can naturally provide certain amount of obfuscation to the control flow, part of the control flow edges are "hidden" inside the cache if blocks are hit in the on-chip cache. For the example in Figure 1.a, if all blocks are in the cache, then the attacker cannot find this loop from outside. Similarly, in Figure 1.b, if both B1 and B2 are in the cache, this control flow edge cannot be detected. However, it does not help due to the following reasons. Firstly, since cache is a shared resource among all processes running on a processor, it is very easy for the attacker to manipulate the OS so that the cache gets flushed on a context switch; alternatively the attacker can either ascertain that his own process fills and occupies most cache space before switching to the process being attacked, or even manipulate the system to disable the use of caches. In this way, all memory accesses are exposed directly on the address bus due to compulsory misses. Secondly, different parts of the control flow can be leaked during different runs. It is possible that the attacker can finally get the whole picture. Finally, on-chip caches are typically small for embedded systems. Most of the modern high-performance DSPs or embedded processors do not even include any on-chip caches for timing predictability for real-time applications.

4. ADDRESS INFORMATION LEAKAGE

In this section, we describe the *address information leakage* problem, which is the objective of hardware-based obfuscation. Our intention is to understand and demonstrate what causes such information leakage. We will also discuss ways to measure the extent of the information leakage.

4.1 Layout Leakage and Recurrence Leakage

When the attacker taps on the bus, he can obtain a sequence of block addresses that are not obfuscated. In Figure 2.a all 5 blocks of instructions are stored in encrypted form, however authentic addresses are readily available on the bus. If address 101 appears after address 100, the attacker can simply infer that these two blocks are executed consecutively. Moreover, the attacker can speculate block 102 will be the *fall-through* path after control flow jumps to 103. Therefore, the original instruction layout in the memory leaks information about which instructions are close to each other on the control flow graph. Notice that, even if the fall-through path to 102 is never executed, the attacker can still reasonably assume such a path exists. We define this kind of information leakage *layout leakage*.

***Layout Leakage**:* Information leakage due to the spatial vicinity of blocks, which implies their proximity of accesses in the control or data flow graph.

Similar to Figure 1.a, the attacker will see 100,101, X, 104 appears repeatedly in the address sequence, where X is either 102 or 103. This gives out the loop and the conditional branch. Essentially, recurring addresses expose loops and branches. We call such leakage *recurrence leakage*.

***Recurrence Leakage**:* Address information leakage due to recurring addresses which indicates the same block is accessed again.

4.2 Address Bus Encryption

DS5000/DS5002FP [5] implements a technique called *address-bus encryption*. Intuitively, since memory contents are encrypted, we can encrypt addresses as well. However, address encryption is radically different. Since we assume the memory is not trusted, once we encrypt an address, the corresponding memory block must be relocated to the new address (i.e. the ciphertext). The following example illustrates that if such encryption is fixed, we still cannot prevent the recurrence leakage.

In Figure 2.c, we show an address bus encryption scheme and Figure 2.b depicts the new memory layout. Notice that, once we encrypt address 100 as 101, the block at address 100 must be moved to address 101 accordingly. The DS5000/DS5002FP processor loads the program which is initially laid out at addresses after encryption. Since the processor knows the encryption scheme, to access a block at address *addr*, it simply issues an access to the address *E*(*addr*). As can be seen, the address bus encryption is actually just a permutation of blocks in memory [5].

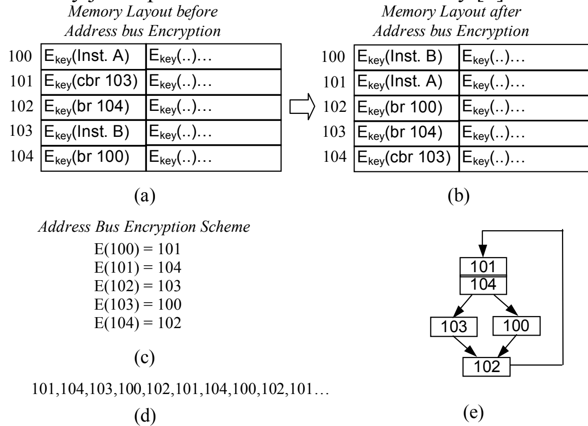

Figure 2. Address-bus encryption.

Clearly, address bus encryption can avoid *layout leakage*, because even sequentially executed blocks are now scattered across non-contiguous memory locations. Address bus encryption involves permutation of blocks in memory, the DS5000/DS5002FP processor never relocates blocks during the dynamic execution, however. Therefore it should be called *fixed address bus encryption* or *fixed permutation*, which means the encryption function never changes. Thus, an address is mapped to a fixed new address and the mapping remains the same throughout the execution.

The implementation of fixed address bus encryption is very simple by providing the mapping inside the processor. On DS5000/DS5002FP, a pseudo permutation function is recorded on chip. All addresses should be mapped by the pseudo permutation function before going out to the bus.

However, fixed address bus encryption completely fails to prevent *recurrence leakage* since the relative access sequence of blocks remains unchanged for the program. In Figure 2.d, we show the address sequence that might appear on the bus. Now, the pattern 101,104, Y, 102 appears repeatedly, where Y can be 103 or 100.

Obviously, the same recurrence pattern can be observed, which means the fixed address bus encryption does not change the recurrence pattern at all, except that each address is mapped to a new address respectively. The new address sequence still recurs in the same manner. In Figure 2.e, the control flow graph can be reconstructed from the address sequence with both loop and the conditional branch exposed.

4.3 Address Recurrences

To further understand the *recurrence leakage* is the critical problem in preventing address information leakage, we consider it in a different way. Assume that the address bus encryption (mapping) is completely random, which means each address is mapped to an independent random address. Also, assume no address is accessed again, i.e. no recurrence occurs. Then the attacker should get a sequence of random numbers on the bus. Therefore, he can extract zero information from the address sequence if no recurrence is generated.

In this paper, we propose approaches to avoid the exposure of address recurrences. Hereby, we define the following different scenarios for address recurrence.

***n-recurrence**:* A series of n recurrences of the same address.

***same-run recurrence**:* Address recurrence in the same run of a program.

***multi-run recurrence**:* Address recurrence across different runs of the same program.

faked recurrence: We can generate faked recurrences, which do not correspond to actual recurrences in the address sequence.

As explained above, address recurrence is the basis for the attacker to extract useful information from the address trace. To conceal address recurrence, i.e., *n-recurrence*, in the original address trace is our main goal. For the address sequence in Figure 2.d, we can observe a 3-recurrence of address 101 and a 2-recurrence of address 100. In section 6, we will show that, with our scheme, the probability of detecting an *n-recurrence* is $\frac{1}{M^n}$, where M is an adjustable system parameter. In other words, it decreases exponentially with n. *Same-run recurrence* can leak control flow information of the code that is executed multiple times during a single run. If the program initial layout is not changed when it starts in each run, *multi-run recurrence* can occur. The attacker can execute the same program many times with different inputs and environments, so the same address may recur at different runs. Even if such addresses do not recur in the same run, information still leaks due to *multi-run recurrences*. In this work, we tackle both *multi-run* and *same-run recurrences*. Besides, in order to confuse the attacker, the processor can generate *faked recurrences* by issuing addresses that have appeared on the bus previously.

4.4 Dynamic Address Mapping and Block Relocation

At this point, we can perceive the insufficiency of fixed address encryption (mapping). Intuitively, we should map an address differently each time it appears on the bus, and then remember the new mapping for future access. However, such a

strategy has to be implemented carefully, otherwise it helps little to prevent leakage. As an example, one naïve implementation could choose to relocate a block immediately after accessing it. For instance, after accessing block 100, we relocate it to 104 immediately. Next time when this block is accessed again, we can get it from address 104. However, since the attacker may watch the bus all the time, he will find 104 is written out immediately after access to address 100 and later 104 is accessed again. Therefore, the attacker has a strong reason to believe that block 104 is actually the original block at address 100. In other words, after the attacker observes address 104 is written out and accessed again, he can speculate an address recurrence. The above example indicates that an effective solution should guarantee little correlation between the relocation operation and the next access. In consequence, the attacker will experience a higher level of difficulty in determining which address the original block is relocated to.

4.5 How to Measure the Security Strength

With a good random permutation (re-mapping) scheme, the new locations of the blocks can be thought as totally independent of their original locations. In this regard, the *layout leakage* can be completely prevented with an initial permutation, because it relies on the relative position of instructions in their original layout.

For *recurrence leakage*, a natural criteria is the probability an *n-recurrence* can be detected by the attacker. The lower the probability of finding an *n-recurrence*, the more difficult for the attacker to guess a loop that reaches its nth iteration. In other words, such probability should be very low, so that the address sequence shown up on the bus is closer to a sequence of random accesses as suggested at the beginning of section 4.3.

5. HARDWARE SUPPORTED CONTROL FLOW OBFUSCATION

Our scheme aims at reducing the probability that *n-recurrence*s are exposed on the address bus. Provably, the scheme we propose can achieve that the probability of an *n-recurrence* being detected is less than $O(1/m^n)$. Here m is a parameter that can be controlled by the amount of hardware resource we are able to invest.

The basic idea for hardware-assisted obfuscation is to relocate blocks every time they are written out to the memory. By doing so, the attacker has to guess from a number of locations in order to find out a possible address recurrence. Our scheme deploys a *shuffle buffer* to achieve this.

Figure 3 shows the main components of our hardware obfuscator. The dotted line cuts the figure into two parts. The upper part is inside the security boundary, i.e. inside the processor chip. The lower part is the bus and memory which are subject to external tapping and tampering. A *shuffle buffer* is added inside the processor chip. As will be addressed in section 5.1, the shuffle buffer relocates blocks to new memory locations. In the meantime, the block address table (section 5.2) records those new locations, so that they can be accessed later. Figure 3 also illustrates how the memory space is divided. The area for storing block address table is only accessible to the block address table caches and the controller. The application cannot see it. The controller accepts requests from the cache, checks with the block address table to find the locations of the blocks and fetches blocks according to their new addresses. Each fetch is always followed by a write to the same location so that the attacker always observes read/write pairs. The controller is also responsible for managing the generation of read-write pairs.

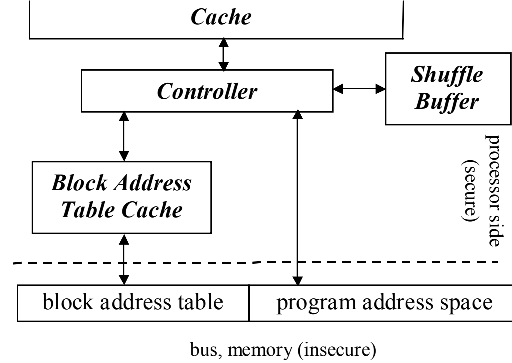

Figure 3. Overview of the hardware obfuscation scheme.

5.1 Shuffle Buffer

The central component for the hardware assisted obfuscation is the *shuffle buffer*. As mentioned before, a block should be relocated to a new location every time it leaves the processor. Also, it should be written out as a part of many other writes instead of being written out immediately. The shuffle buffer is designed to reorder all writes to the memory so that the attacker cannot easily determine exactly which one is written out at a particular moment. Write reordering can significantly mask address recurrences as will be formulated and quantified in section 6.

The structure of the shuffle buffer together with part of the functionalities of the controller is shown in Figure 4. There are 3 data paths centered around the shuffle buffer. Here, we assume the on-chip cache is a write-back cache.

The shuffle buffer is simply an array of blocks. Each code block or data block is either found in the memory or in the shuffle buffer but not in both of them. In this sense, the shuffle buffer is exclusive to the external memory in contrast to the cache hierarchy, which is inclusive. From Figure 4, data path (1) and (2) are similar to those in the caches. Cache misses are handled by the controller. For cache misses that hit in the shuffle buffer, blocks are immediately fetched from the shuffle buffer to the cache. If the cache miss is not found in the shuffle buffer, we have to fetch it from the memory. Upon finishing the memory access, both the shuffle buffer and the cache will get a copy of the block.

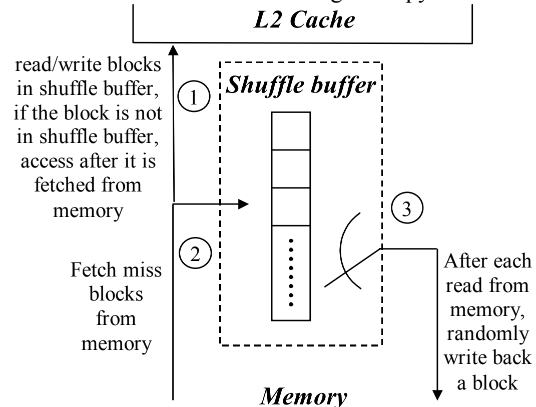

Figure 4. The shuffle buffer.

However, there are several distinct properties for the shuffle buffer in addition to the design that a block can only stay in either

the shuffle buffer or the memory. The shuffle buffer is simply an indexed array through the block address table, thus no address tag is necessary. The size of the shuffle buffer can be much smaller than the cache, making our scheme inexpensive. To determine if a cache miss is a hit in the shuffle buffer or not, the controller relies on the block address table which records the whereabouts of blocks. The shuffle buffer allows any block to be stored in any slot. Moreover, once a block is fetched into the shuffle buffer, it can replace a randomly picked block inside the shuffle buffer. The above property can also be achieved by employing a fully associative cache, but it is more expensive and less scalable. Since we need block address table to record the locations of the blocks in the external memory, it is natural to use block address table to record the location of blocks in shuffle buffer too. Thus, no fully associative organization and tag comparisons are needed for the shuffle buffer.

As aforementioned, an incoming block can replace any existing block in the shuffle buffer. As shown with data path (3), the controller picks a block randomly from the shuffle buffer to write back to memory. Here we assume a pure random number generator is available using circuit white noise [20]. Thus, the controller can pick a completely random block in the shuffle buffer to be replaced by the incoming block and write back the replaced block to the same address of the incoming block.

It is interesting to see that the number of blocks in the shuffle buffer is always fixed, since we always write out a block to the address of the fetched block, immediately after a block is fetched in. In this way, the write-out block is simply put into the memory location where the incoming block is from. Dynamically, the number of blocks that are in the shuffle buffer does not change. Initially, we could load part of the memory blocks to fill the shuffle buffer before the program starts execution. After loading, these memory blocks are in the shuffle buffer but no longer exist in the memory. Once the program stops execution, everything inside the shuffle buffer is written back to the space where the initially loaded blocks come from. Assume that the program binary can be updated, then *multi-run recurrences* is prevented naturally, because during the execution of the program, the blocks are relocated frequently and after the execution of the program, we get a new permutation of the blocks in memory.

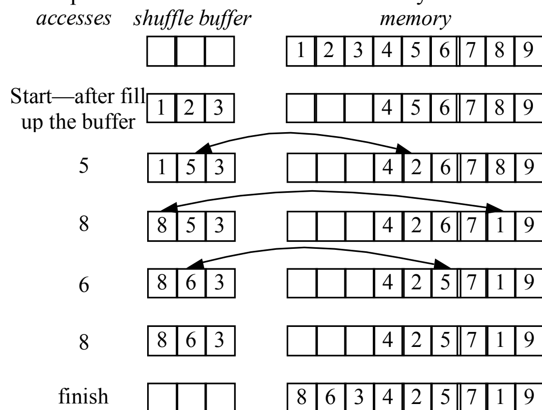

Figure 5. Example for the shuffle buffer.

To illustrate how the shuffle buffer functions, an example is shown in Figure 5. Assume the code contains 9 blocks. In this example, we only look at code blocks and we will discuss data blocks shortly. At the beginning of this run, the first three code blocks are loaded into the shuffle buffer. This leaves three empty block slots in the memory, and they will not be used through the execution. When block 5 is accessed, the controller checks with the block address table and finds out block 5 misses in the shuffle buffer, therefore the controller issues a memory access. Upon loading block 5, the controller also randomly picks a block in the shuffle buffer (say block 2) to be replaced from the shuffle buffer and writes it into the original location of block 5. When block 8 is accessed, the controller picks block 1 to be replaced. The selection of blocks to replace is totally random. After block 6 is loaded, block 5 is chosen to be written out and put in the original place of block 6. The next access to block 8 hits in the shuffle buffer, therefore no memory access takes place. Finally, when the program finishes this run, all blocks in the shuffle buffer are written into the first three empty blocks in the memory. The blocks in memory are in random order compared with their initial locations before the execution.

Again, the controller can always learn from the block address table to get the exact location of each block. When the shuffle buffer is sufficiently large, a block may leave the shuffle buffer at a random moment spanning across a long period. Moreover, since all the blocks are re-encrypted upon eviction (the stream cipher encryption/decryption [11][12] is very efficient for this purpose), the attacker will not be able to correlate read-in blocks and written-out blocks from the contents of the blocks. The above two points make it very difficult for the attacker to identify any recurrences. We will analyze the security strength of the shuffle buffer scheme in section 6.

The example in Figure 5 illustrates one way to avoid *multi-run recurrences*. It relies on the assumption that after one run of the program, we can write back the reordered code to the disk. Consequently, every run will start with a different layout of the code. The same block accessed in the previous run must have been written to a different location, when it is accessed again in the current run. However, this would require OS support to allow write-back of modified code (i.e. self-modifying code), since normally program code is read-only. Moreover, the block address table has to be stored along with the code after a program terminates. Another way to prevent *multi-run recurrence* is to enforce an initial random permutation when the program is loaded from the disk. A temporary block address table is constructed in memory to store block locations. After the program terminates, the temporary block address table is flushed.

As addressed in [15], data address can disclose control flow information as well. For instance, when a branch is taken, variable x will be accessed, otherwise there is no reference to x. Consequently the attacker can learn whether the branch is taken or not by checking the access to x. For data blocks, we can handle them in the same way as code blocks.

5.2 Block Address Table (BAT) and Block Address Table Cache (BAT cache)

Block address table (BAT) records the current locations of blocks. To find the current location of a block, we can use its original block address to index into the BAT and find out its new block address. We map the shuffle buffer into the virtual address space of the application (the program itself is not allowed to touch this part), so the new block address in the virtual address space can tell whether the block is in memory or in the shuffle buffer. Simply put, for each block we need to store a block address, which

is an address less the block offset bits. To access a code or data block, we use its block address to index into the BAT, i.e., we access the entry at *BAT.base_addr + block_address * BAT.entry_size*. If we assume all addresses are 32 bit long and the block size is 32 bytes, then the block address field is 27 bits, resulting in a 10 % (=27/256) overhead in virtual memory space. Note that this is the worst case estimation. If the program actually takes less memory space, this overhead can be reduced. For example, if we know the actually address space is less than 2^{24} bytes, i.e. 16MB. With a 32-byte block size, the overhead is 7% (=19/256).

Since each access request from the cache needs to be checked with the BAT table, the latency to retrieve entries of the BAT can slow down the critical reads. To speed up the accessing of the BAT, we add a *block address table cache (BAT cache)* as shown in Figure 3. The BAT cache is typically small due to the small size of the BAT, since it only takes a small space overhead (e.g. 6%) to accommodate the BAT, which is generally sufficient to yield a high hit rate. Also, the hit latency of the BAT cache is very low. Under our scheme, the misses from BAT cache are satisfied by the data cache.

A data cache miss caused by a BAT cache look up will not look up BAT cache again. Instead, it will get data from memory directly. Potentially, there is information leakage due to BAT data accesses. However, one block of BAT data can cover at least 32*9 bytes instructions in our processor model as shown above, thus the leakage due to BAT cache accesses is minor. To build a hierarchical protection scheme in which a cache miss due to a BAT access will look up BAT cache again is another choice, which achieves better security but the performance overhead is bigger at the same time. In our work, we choose the first scheme.

6. SECURITY ANALYSIS

We now analyze the security strength of our scheme. For the attacker to find out a recurrence, he or she has to determine when (thus where) a previous read-in block is written out. Figure 6 shows read/write *access pairs* on the bus. As mentioned earlier, every time a block is read in, we write out one block immediately to the same location. Assume a block is read in during access pair A_0, since in each of the following access pairs, we write out a block randomly from the shuffle buffer. If we select each block in the shuffle buffer with the same probability and the size of the shuffle buffer is M blocks, then the probability that the read in block at A_0 will be output at A_k is $P_k = (1-\frac{1}{M})^{k-1} \times \frac{1}{M}$. This probability monotonously decreases with k and the maximal value, i.e. $\frac{1}{M}$, is reached when k=1. Therefore the best choice for the attacker is to guess the block is written out at A_1. In our experiments, M is set to 128 by default, which means the attacker only has 0.8% chance to guess one recurrence correctly. For *n-recurrence*s, as each random selection is independent, the chance for the attacker to guess all correctly is $\frac{1}{M^n}$.

Obviously, we can achieve better security with larger M. However, a larger M leads to more on-chip space investment and bigger overhead during context switches. Notice that, with our current configuration, i.e. M=128 and 32B block, only 4KB is required for each shuffle buffer. Moreover, a shuffle buffer is cheaper than caches because of its lack of a tag array. Given that modern IC fabrication technology provides us with ample on-chip space, this overhead is less a concern.

Read/write access pairs on the bus

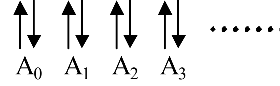

Figure 6. Access pairs on the bus.

7. OTHER CONSIDERATIONS

Dynamically Linked Libraries (DLLs) are inserted into a program's address space at runtime. A DLL can be handled just like a program. Since a DLL might be shared by multiple applications, the shuffling should be done in a centralized way to maintain correctness.

In-between each context switch, all the on-chip data private to the current process including all the shuffle buffers for the process must be cast out. The overhead for our schemes does not increase the overall cost significantly given the on-chip structures are typically small. The performance impact in a multi-task system is currently open.

Table 1. Default configuration.

Processor		Cache/Memory	
Clock frequency	200MHz	L1 I/D	32 way, 32K, 1 cycle
Fetch queue	8 entries	L1 D ports	1
Decode width	1	L1 latency	hit 1 cycle,miss 32 cycles
Issue width	2	TLB miss	30 cycles
Commit width	2	HW obfuscator	
Ruu size	4	BAT cache	1KB
Lsq size	4	Shuffle buffer	128 entries
Branch predictor	no		

8. Evaluation and Results

We evaluated 12 benchmark programs from Mibench [19]. The Mibench suite, quite close to the industrial standard EEMBC benchmark suite, is freely available. The selection of benchmarks are completely random. To cover the whole benchmark suite properly, we include two benchmarks from each of the six categories--qsort, susan (auto/industry), jpeg, lame (consumer), dijkstra, patricia (network), ispell, rsynth (office), blowfish, rijndael (security), adpcm, gsm (telecom).

We evaluate our scheme on a processor model with default parameters in Table 1, which largely follows an ARM processor model in Simplescalar [18]. In our experiments, we implemented the one time pad (OTP) encryption, a stream cipher based scheme used in [11][12]. The advantage of the OTP scheme is the fast encryption/decryption speed with a low overhead. By default, the shuffle buffer has 128 entries, BAT cache size is 1K.

Under our scheme, the shuffle buffer itself does not degrade performance. On the other hand, since the shuffle buffer resembles the functionality of a victim cache, it actually improves performance in some cases. The degradation under our scheme is primarily caused by the additional access to the BAT. Before the fetch of a missed block, the BAT has to be accessed first to get the current location of the block, procrastinating the delay on the critical path. To alleviate the performance impact, the active part of the BAT should be kept on-chip for reducing the access latency. That is the purpose of implementing a separate BAT cache.

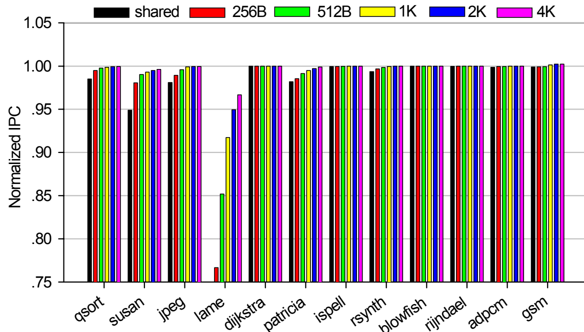

Figure 7. BAT cache sensitivity study--IPC.

Figure 7 shows the sensitivity study with a variety of the BAT cache sizes. Using the default configuration, we varied the BAT cache size from 256 bytes to 4KB. We also studied the case when no separate BAT cache is present and all accesses to the BAT are routed to the cache directly. The IPC numbers are normalized to the baseline (i.e. no hardware obfuscation). On average, with shared cache, the degradation due to hardware assisted obfuscation is 3.6%; with a 256B separate BAT cache, the degradation drops to 2.4%; with a 512B BAT cache, it is reduced to 1.5%. With the size of the BAT cache continuing to increase to 1KB, 2KB and 4KB, the overall performance degradation is further shrunk down to 0.8%, 0.5%, and 0.3%, respectively. As shown from the results, the BAT cache is very effective to close the performance degradation, especially for benchmarks susan, jpeg and lame. Without a separate BAT cache, all the BAT accesses will otherwise go to the cache, leading to some severe pollution for those benchmarks with a relatively large working set. With a bigger BAT cache, most BAT accesses hit in the BAT cache, reducing the performance degradation, however we observe the return is diminishing when the size is over 1KB.

Figure 8 shows the hit rate of BAT cache with the BAT cache size varied. It is clear that the hit rate increases rapidly with a larger BAT cache. Also, the hit rate of a small BAT cache can be very low for some benchmarks. This is because the BAT accesses are triggered and indexed by cache misses. When the cache demonstrates poor locality, the accesses to the BAT will be lack of locality too. On average, the hit rates are 61.7% for a 256B BAT cache, 75.9% for a 512B BAT cache, 87.5% for a 1KB BAT cache, 92.9% for a 2KB BAT cache, and 94.1% for a 4KB BAT cache.

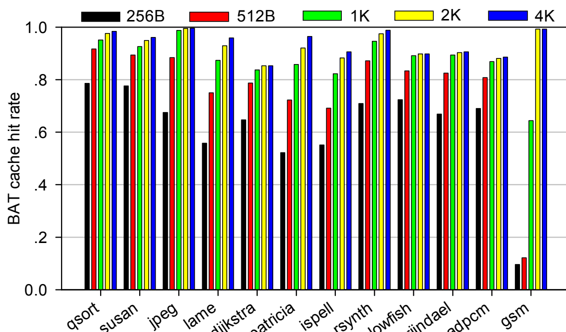

Figure 8. BAT cache sensitivity study--hit rate.

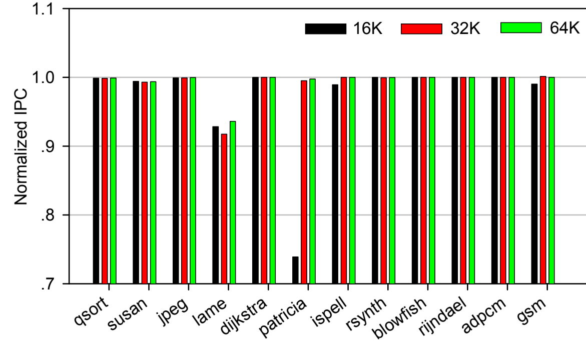

Figure 9. Cache size sensitivity study.

Figure 9 studies the sensitivity with respect to the (unified I/D) cache size when all other parameters remains the same as in the default model. The IPC numbers are normalized to that of the baseline (i.e. no hardware obfuscation). A larger cache is more tolerable to the pollution caused by the BAT accesses, so the performance degradation due to our hardware assisted obfuscation should be smaller, which is confirmed in Figure 9. For a 16KB cache, the average degradation is 3.1%. For a 32KB cache, it is 0.8% while it drops down to 0.6% for a 64KB cache.

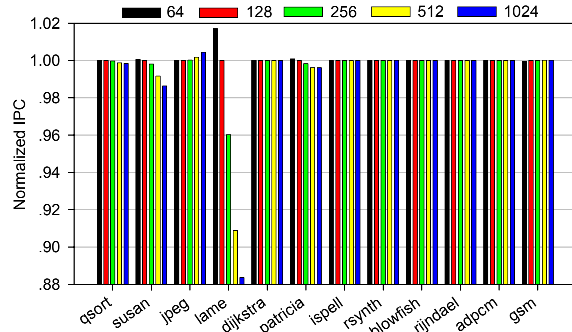

Figure 10. Shuffle buffer size sensitivity study.

Figure 10 shows the sensitivity study of the shuffle buffer size. Only the shuffle buffer size is varied in the default model and IPCs are normalized to that of the default model. It may be surprising to see that generally, a larger shuffle buffer leads to a little worse performance. We argued earlier that a shuffle buffer can function like a victim cache thus has the effect to reduce the penalty of cache misses. Some benchmarks do show the victim cache effect, e.g. jpeg, rsynth. Next, we discuss the negative effects of a larger shuffle buffer on performance. As explained earlier, whenever there is a block fetched into the shuffle buffer, the shuffle buffer has to randomly pick a block to replace. For each read/write pair, there are two accesses to the block address table. The first access is to look up the current location of the missed block and to fetch it in. The second access is a write access to update the block location for the replaced block, which is written into the current location of the fetched block. The write access is affected by the shuffle buffer size. With a larger shuffle buffer, the write accesses will have less locality, leading to more BAT cache misses and more pressure on the cache. To summarize, on average shuffle buffer sizes of 64-entry, 128-entry and 256-entry yield the same performance. A 512-entry shuffle buffer

causes 0.9% slowdown and 1024-entry shuffle buffer leads to 1.1% slowdown. Nevertheless, a larger shuffle buffer will provide a stronger security guarantee as analyzed in section 6.

Besides the above performance evaluation, we also performed experiments to evaluate the effectiveness of our hardware obfuscation. We downloaded the random number generator evaluation tool suite released by NIST and applied a battery of statistical tests [22] to evaluate the randomness of the address trace generated. Before we show the analysis results, we must point out that the goal of our hardware obfuscator design is not to truly randomize the instruction address stream, rather, the obfuscator is designed to obscure the control flow of a program and confuse the potential adversaries. If the shuffled address trace demonstrates higher randomness than the original address trace, it is simply a positive second-order effect of our obfuscator. In some cases, the shuffled address trace cannot be very random since the size of the shuffle buffer is limited and during a certain period, the addresses accessed by the program are concentrated. It was also observed that the obfuscator did not pass in some tests designed for evaluating entropy (test "apen" in the tables below) and compressibility (tests "universal" and "Lempel-Ziv" in the tables below)". The reason is that he goal of our design is not to reduce entropy or compressibility, the calculation of which are both based on *frequency*. On the other hand, the obfuscator focuses on manipulate *sequence* rather than *frequency*. Thus, the results from the randomness testing tool suite should be evaluated relatively. We compared the randomness testing results of our hardware obfuscator against a pure cache design, which also obfuscates address sequence to some extent. From the results below, it should be clear that address trace from our hardware obfuscator is much more random than that from a cache.

To generate the bit stream needed for randomness testing, we run the same 12 embedded programs used in performance evaluation from MiBench for 300 million instructions and collect address traces during the execution. Each address going through memory bus is first shifted right to remove the trailing 0s caused by cache line size boundary, then the rightmost 16 bits of the address are dumped as the address trace for analysis. Without such processing, the leading and trailing 0s will make the sequence not random at all.

There are totally 16 randomness tests in the tool suite. We show the results for 12 of them below. We did not perform template matching tests since the results are dependent on the template. We did not perform excursion tests since the tool suite does not generate correct results due to underflow errors. We evaluated the address traces for three configurations: 8K I-cache and D-cache, 16K I-cache and D-cache, and our shuffle buffer scheme without cache at all. The shuffle buffer evaluated has 128 entries. Cache and shuffle buffer combined schemes do not improve randomness according to our experiments thus are not presented. With same amount of instructions executed, the size of the address trace generated depends on the configuration. The configuration with a single shuffle buffer will have the largest trace. Among the 12 benchmarks, the address traces generated by some of them under 16K cache configuration are too short for randomness testing, i.e., less than 1 million bits. Those benchmarks are excluded. Finally, we show results for 7 benchmarks with reasonable large address traces.

The tool suite takes an address trace as input and splits the trace into multiple binary sequences having same length. Each binary sequence is tested against each randomness test once. The length of the sequence should be large enough to get reasonable results. In our experiments, the length is 1 million. Each table below shows the randomness testing results for one benchmark. In the table, the total number of sequences (the total number of tests) in the address trace of the benchmark is shown under the configuration. For each test, the number of sequences passing the test is shown in the corresponding entry. We also experimented with a "no cache no shuffle buffer" configuration, and all the randomness tests returned "0" results, i.e. all sequences failed to pass any test, showing there is no randomness in the address stream for a bare bone system without cache. Please refer to [22] for the details of the tests.

Table 2. Randomness testing – cjpeg.

tests	8K	16K	SB
	4	2	8
Frequency	0	0	1
Block-Frequency	0	0	0
Cusum 1	0	0	0
Cusum 2	0	0	0
Runs	0	0	1
Long-Run	0	0	0
Rank	0	0	0
FFT	0	0	6
Universal	0	0	0
Apen	0	0	0
Serial 1	0	0	0
Serial 2	0	0	1
Lempel-Ziv	0	0	0
Linear-Complexity	2	2	8

Table 3. Randomness testing – gsm.

Tests	8K	16K	SB
	21	10	27
Frequency	0	0	16
Block-Frequency	0	0	8
Cusum 1	0	0	14
Cusum 2	0	0	16
Runs	0	0	6
Long-Run	0	0	3
Rank	0	0	19
FFT	0	0	27
Universal	0	0	20
Apen	0	0	0
Serial 1	0	0	3
Serial 2	0	0	6
Lempel-Ziv	0	0	0
Linear-Complexity	21	10	27

The above results demonstrate two points clearly. First, cache is unable to obfuscate the address trace effectively. Under a pure cache scheme, few binary sequences pass any randomness test. Linear-complexity test is a corner case. Moreover, increasing the cache size from 8K to 16K does not improve the situation. Second, our hardware obfuscate does a much better job than a pure cache design. Under our hardware obfuscator scheme, the frequency of binary sequences passing randomness tests is much higher. It is obvious that the hardware obfuscator can generate more random address traces even though it is not our first-priority design goal.

Table 4. Randomness testing – ispell.

Tests	8K	16K	SB
	19	3	171
Frequency	0	0	32
Block-Frequency	0	0	0
Cusum 1	0	0	1
Cusum 2	0	0	5
Runs	0	0	11
Long-Run	0	0	3
Rank	0	0	1
FFT	0	0	15
Universal	0	0	0
Apen	0	0	0
Serial 1	0	0	9
Serial 2	0	0	28
Lempel-Ziv	0	0	0
Linear-Complexity	19	3	157

Table 5. Randomness testing – lame.

Tests	8K	16K	SB
	110	59	227
Frequency	0	0	54
Block-Frequency	0	0	0
Cusum 1	0	0	19
Cusum 2	0	0	23
Runs	0	0	17
Long-Run	0	3	12
Rank	0	0	1
FFT	0	0	192
Universal	0	0	0
Apen	0	0	1
Serial 1	0	0	8
Serial 2	0	0	38
Lempel-Ziv	0	0	0
Linear-Complexity	109	59	193

Table 6. Randomness testing – qsort.

Tests	8K	16K	SB
	4	3	437
Frequency	1	0	39
Block-Frequency	0	0	19
Cusum 1	0	0	13
Cusum 2	0	0	14
Runs	0	0	8
Long-Run	0	0	42
Rank	0	0	355
FFT	0	0	239
Universal	0	0	37
Apen	0	0	0
Serial 1	0	0	5
Serial 2	0	0	51
Lempel-Ziv	0	0	0
Linear-Complexity	4	3	426

9. RELATED WORK

As mentioned earlier, secure architectures [6][7][8][9][10] that solely encrypt memory contents cannot change the addresses sequence, resulting in the unobfuscated control flow to be leaked entirely. The DS5000/DS5002FP series processor [5] features address bus encryption, equivalent to the initial permutation in our scheme. However, it does not permute repeatedly at runtime, therefore the attacker can still reconstruct the CFG in the same way as mentioned in section 4.2. The DS5000 also issues random fetches in order to generate *faked recurrence* to confuse the attacker. However, random fetches can be easily discerned from true accesses in loops, since true accesses repeat much more frequently. Actually, DS5002FP has been completely cracked [14].

Goldreich [16][17] proposed 3 approaches to guarantee no address information leakage, however all of them can incur intolerable slowdown or memory space overhead. For example, the "square-root solution" needs to read the entire shelter buffer before each access; the "hierarchical solution" takes O(t*log(t)*log(t)) memory space after t accesses, causing memory explosion. Therefore these approaches are not affordable even for high end systems, let alone for embedded systems.

Table 7. Randomness testing – rsynth.

Tests	8K	16K	SB
	27	2	80
Frequency	0	0	1
Block-Frequency	0	0	0
Cusum 1	0	0	0
Cusum 2	0	0	1
Runs	0	0	0
Long-Run	0	0	1
Rank	0	0	0
FFT	0	0	70
Universal	0	0	0
Apen	0	0	0
Serial 1	0	0	0
Serial 2	0	0	4
Lempel-Ziv	0	0	0
Linear-Complexity	27	2	70

Table 8. Randomness testing – patricia.

Tests	8K	16K	SB
	377	320	588
Frequency	0	0	124
Block-Frequency	0	0	257
Cusum 1	0	0	119
Cusum 2	0	0	115
Runs	0	0	40
Long-Run	0	0	147
Rank	0	0	424
FFT	0	0	497
Universal	0	0	135
Apen	0	0	2
Serial 1	0	0	30
Serial 2	0	0	124
Lempel-Ziv	0	0	0
Linear-Complexity	366	313	497

10. CONCLUSION

This paper presents a hardware assisted obfuscation technique to obfuscate the control flow. We show that encryption alone cannot avoid the leaking of control flow information. The control flow leakage can imperil both code and data encryption. Traditionally, some software obfuscation techniques can transform the program control flow to reduce such information leakage. However, a pure software-based obfuscation has been proved inefficient to protect software IP due to its lack of theoretical foundation and considerable performance overhead introduced by complicated transformations.

Obfuscation with hardware support as introduced in this paper can achieve a high level of security guarantee incurred with

very low overhead, making it feasible for embedded processors. It is shown that the chance for the attacker to identify n-recurrence decreases exponentially with n, therefore it is extremely hard to extract useful information from the address sequence after hardware obfuscation.

Our experiments on an embedded processor model show that, the performance degradation can be below 1%. The hardware cost consists of small on-chip shuffle buffers and a BAT cache. For low-end embedded systems with limited hardware budget and severe control flow leakage (due to small caches), this overhead is affordable, making our scheme necessary and feasible. Moreover, through standard tests to evaluate randomness, we show our hardware obfuscator is much more efficient than a pure cache design in terms of making address trace random.

REFERENCES

[1] International Planning and Research Corporation, "Eighth Annual BSA Global Software Piracy Study," *http://global.bsa.org/globalstudy/2003_GSPS.pdf.*

[2] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, 1997.

[3] Chenxi Wang, "A scecurity architecture for survivability mechanism," PhD Dissertation. Univ. of Virginia, Department of computer science, Oct. 2000.

[4] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)possibility of Obfuscating Programs. In J. Kilian, editor, Advances in Cryptology -- CRYPTO '01, pages 1--18.J.

[5] "DS5002FP secure microprocessor chip data sheet," *Dallas Semiconductor*.

[6] Markus, Kuhn, "The TrustNo 1 Cryptoprocessor Concept," CS555 Report, Purdue Univ. 1997.

[7] Robert M. Best. Preventing software piracy with crypto-microprocessors. In Proceedings of IEEE Spring COMPCON 80, page 466, February 1980.

[8] Kent, S.T, "Protecting Externally Supplied Software in Small Computers," Ph.D Thesis, MIT/LCS/TR-255 1980.

[9] White, Steve R.; Comerford, Liam: ABYSS: A Trusted Architecture for Software Protection," *Proc. 1987 IEEE Symposium on Security and Privacy*, Apr. 1987.

[10] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," *ASPLOSIX'00*.

[11] Yang, Y.Zhang, L.Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering," *MICRO'03*.

[12] E.Suh, D.Clarke, B.Gassend, M.v.Dijk, S.Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors", *MICRO'03*.

[13] B.Gassend, G.E.Suh, D.Clarke, M.v.Dijk, S.Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification", *HPCA9*.

[14] M.G.Kuhn, "Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP," *IEEE Trans. on Computers*, Vol.47,No.10, pp.1153-1157, 1998.

[15] X. Zhuang, T. Zhang, S. Pande, H.S. Lee, "HIDE: Hardware-support for Leakage-Immune Dynamic Execution," GIT-CERCS-03-21.

[16] O.Goldreich, "Towards a Theory of Software Protection and Simulation by Oblivious RAMs," *Proceeding of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, 1987.

[17] O.Goldreich, R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *J. of the ACM*, Vol.43,No.3, 1996.

[18] Doug Burger and Todd M. Austin. "The SimpleScalar Tool Set Version 2.0," TR. 1342, Univ. of Wisconsin--Madison, May 1997.

[19] M.R.Guthaus, J.S.Ringenberg, D.Ernst, T.M.Austin, T.Mudge, R.B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[20] Intel Corporation, "Intel Random Number Generator," http://developer.intel.com/design/chipsets/rng/techbrief.pdf, 1999.

[21] X. Zhuang, T. Zhang, S. Pande "HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus", *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-04),* Oct. 2004.

[22] National institute of standards and technology, "A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications," available at http://csrc.nist.gov/rng/SP800-22b.pdf.