HIDE: Hardware-support for Leakage-Immune Dynamic Execution

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

Georgia Institute of Technology
College of Computing
801 Atlantic Drive
Atlanta, GA, 30332-0280
{xt2000, zhangtao, santosh}@cc.gatech.edu *leehs@ece.gatech.edu

Abstract

Secure processors have been recently introduced, which enable new applications involving software anti-piracy, program execution certification, and secure mobile agents. Secure processors have built-in hardware support for cryptographic mechanisms and can prevent both software attacks and physical attacks. Several recent papers have shown how to construct a secure processor to protect the confidentiality [1][2][3]and integrity[4][3] of a program. The proposed designs are immune from spoofing, splicing and replay attacks. However, none of the previous work is able to address the attacks due to information leakage on the address bus. Dangers due to information leakage on the address bus have been acknowledged to be an important as well as a difficult problem[1]. In fact, in [4]this problem is actually the trigger of the replay attack described.

In this paper, we show that several attacks are possible by monitoring the instruction access sequence on the address bus. Such attacks could emanate from identifying the core algorithms by pattern matching the control flow graph or from finding out or narrowing down critical variables that decide outcomes of conditional branches. We analyze the causes behind such information leakage and then determine the primary requirement that must be met to prevent it. Based on this requirement, we propose HIDE, a hardware-based approach to hide the instruction access sequence. The main goal of HIDE is to construct a fixed instruction access sequence issued to the memory to achieve zero leakage of control flow information, giving a security guarantee. Our base approach involves constructing a fixed instruction access sequence covering the whole program (called base access ring) to hide the actual instruction fetch. This might however lead to severe performance degradation due to tremendous stalls making the framework infeasible. Therefore, we propose two approaches to overcome this problem. In our scheme, the architecture dynamically tracks a hot function set. Based on the hot function set, the first approach involves prefetching blocks accordingly into an on-chip prefetch buffer. The second approach establishes a secondary access ring, which is smaller and faster than the base access ring. The instruction blocks are prefetched from the base ring into the secondary ring instead.

We observe considerable elimination of degradation due to our architectural improvements. For 512K L2 cache, the degradation is reduced from 73% to 38%; for 1M L2, it is cut from 65% to 34% with a reasonable amount of hardware resource.

Keywords:

Leakage Protection, Secure Processor, Instruction Access Sequence

1. Introduction

Software rights protection is a very important issue faced in today's software industry due to billions of dollars invested in the intensive software development process. The idea of software rights protection is to protect the intellectual property (IP) that forms a basis of the software. Use of such underlying intellectual property must be paid for in terms of licensing fees and its use should only be limited to an environment for which a license is granted. Most of the current violations of these rights manifest themselves in terms of illegal copies. Commonly, software copy protection or software piracy protection, which aims to prevent one from making an illegal copy of the software, is touted as a solution to solving the bigger problem of software rights protection. Piracy itself has been a critical and extremely difficult challenge faced by software vendors. According to the study done by the Business Software Alliance, global dollar loss due to software piracy increased 19% in 2002 to $13.08 billion [13], reflecting the severity of the problem. Apart from piracy, the bigger problem of rights is even severe. Hackers often get *illegal insights* into the working of software, discover vulnerabilities and launch attacks. The damage done by the attacks far exceeds the costs of piracy alone. Thus, the problem of software rights is becoming more and more important.

On the other hand, traditional software copy protection techniques like serial numbers and software licensing systems are becoming weaker and weaker with the presence of powerful disassemblers and debuggers. The failure of traditional software

copy protection schemes resides in the fact that they try to protect software by software. Since any software can be reverse engineered, such schemes can only deter the success of an attacker.

Recently, secure processors have been proposed [1][2][3][4][5][14] as a hardware solution to software copy protection, in which a hardware security boundary is demarcated. Anything inside the security boundary is trusted, while the other parts of the computer system are not trusted or in other words can be fully manipulated by the attacker. Normally, components such as processor itself and on-chip caches are inside the security boundary. On the other hand, components such as external memory or I/O devices are outside the security boundary. Both code and data going out of the security boundary must be encrypted. Lots of efforts have been put to ensure the secure processor sustain from various attacks such as spoofing, splicing, replay [1][4] and to reduce the runtime performance penalty due to additional cryptographic operations [2][3].

Encryption is one of the most powerful techniques used to prevent making illegal copies directly. However, Goldreich and Ostrovsky [8] pointed out that software protection cannot be achieved through mere encryption, any information regarding to the software must be prevented from being leaked out during the execution of the software. As pointed out earlier, such an information leakage can lead to intellectual property being stolen and the use rights granted being violated. For example, side channel information such as timing or power consumption of operations can all be utilized to discover the vulnerabilities of the software and launch attacks. The dangers are very real and many commercial smart cards have been cracked by exploiting the information leaked when they are operating [18][19]. Similarly many operating systems vulnerabilities have been discovered by observing and tampering critical data locations (such as return addresses in stack etc.). Thus, leakage protection is critical yet definitely more difficult to achieve. Information leakage at runtime may reveal dynamic control flows, sensitive data values and other valuable information that might be exploited by the attacker. In many cases, it easily invalidates copy protection through encryption.

In this paper, our focus is to combat the leakage of *instruction access sequence* on the address bus. Section 2 shows, although the attacker has no plaintext of the code after encryption, the instruction access sequence can actually help him to guess a lot of information about the code indirectly. In other words, without such a leakage protection, software copy protection through encryption is not sufficient. Moreover, sensitive data can be leaked as well. Actually, the problem of information leakage via instruction access sequence has been acknowledged in both academia and industry. Goldreich [7][8] studied the problem of hiding program memory access pattern from a theoretical point of view. However, their model is no longer proper for modern processors and under their proved lower bound, program performance may be degraded tens of times, making the scheme unrealistic. XOM based approaches encrypt code and data outside the security boundary, making direct software piracy almost impossible. However, it fails to address the information leakage through instruction access sequence. This greatly decreases the strength of copy protection as shown in section 2. Although the problem has been noticed in [1] and [5], they both leave it open. [1] poses it as an open problem, and [5] largely ignores it. In [4], the detection of loops through the address bus becomes a starting point for the replay attack. DS5002FP microcontroller [9] is a widely used commercial *bus-encryption* processor. In DS5002FP, code blocks are stored in new addresses, which are the encrypted values of their original addresses. However, this simple bus encryption does not stop control flow leakage as well as explained in related work section.

In summary, current architectures do not provide a satisfactory solution to the leakage problem we attempt to solve.

In this paper, our contributions are:

1. We show practical attack methods that exploit information from the unprotected instruction access sequence. Leakage from such sequence is so severe that the effects of the code and data encryption get nullified to certain extent.
2. We propose HIDE, an architecture approach, which *completely* eliminates information leakage due to the instruction access sequence, while the performance degradation is still within a tolerable range.

The rest of the paper is organized as follows: Section 2 discusses the motivation; section 3 deals with the preliminaries; section 4 discusses independent access sequence; section 5 presents HIDE in detail; section 6 shows evaluation results; section 7 discusses about related work and section 8 concludes the paper.

2. Motivation

This section elaborates two kinds of attacks through the instruction access sequence disclosed on the address bus. We assume both code and data are protected by encryption such as in XOM[1], DS5002FP[9], and Goldreich's model[7][8], therefore cannot be obtained directly, but the instruction access sequence is subject to outside tapping. We will show that reuse code, which takes about 39% of all code in the benchmark, can be easily identified. Based on that, sensitive data is vulnerable due to value-dependent conditional branches, making both code and data encryption ineffective.

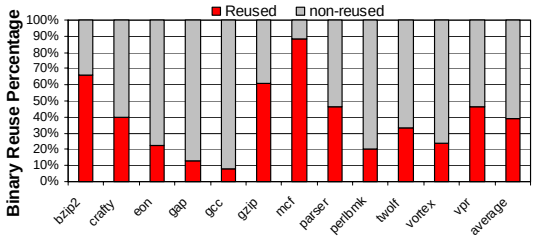

Figure 1. Binary reuse percentage for SPEC2000.

For a better understanding, we will first discuss the worst case when the cache is not present like in many embedded processors e.g. some DSP, smartcard chips. Then we point out that even when caches are enabled, they cannot completely prevent information leakage on address bus.

Without caches, one obvious observation is that the dynamic control flow of the software is completely exposed to the attacker. Although the attacker has no idea about the actual instructions being executed (all instructions are encrypted), he can still obtain control flow information like branches and loops. Control flow contains the most fundamental information of a program and it provides sufficient details for the attacker to understand the program. The scenarios described in this section are only two possibilities.

2.1 Reuse Code Identification

It is highly possible that the attacker can eventually capture most of the control flow after monitoring the address bus for a sufficient amount of time and experimenting the program with different inputs, because theoretically, only dead code will not be executed. Due to the following two facts, leaking the program control flow information can result in the exposure of reuse code and severely disrupt code encryption, even threaten the program's intellectual property.

Software Reuse and Binary-Level Similarity

With the ever increasing of software complexity and time-to-market pressure, software development more and more relies on reusing existing modules or libraries from other companies or oftentimes from the public domain. For example, lots of classic algorithms have their standard and/or non-standard open-source implementations online. To save time, programmers tend to simply reuse the available implementations rather than crafting one from scratch. Moreover, most compilers and development tool chains are provided by a few 3rd party name-brand vendors. As a result, as long as code is reused at source level, they are similar at binary-level. In reality, most reuses are through standard prebuilt libraries, leading to nearly identical code at binary-level. We measured all SPEC 2000 Alpha binaries to find out the percentage of code that is reused from the standard C library on Alpha. As shown in Figure 1, the reuse percentage can be very high for some benchmarks like mcf (88%) and bzip2 (66%). On average, 39% of the code at binary level is from the libraries. Notice that, programs like gcc have been developed for over 10 years and very few existing modules are incorporated from older versions and thus, it is not surprising its reuse percentage is lower. However, with the exploding number of legacy code and shortened software generation time, the reuse code percentage should be much higher for most software on the market nowadays.

CFG-Fingerprint of Algorithm

It is widely known that the average length of basic blocks is only 5 to 8 instructions and a large number of instructions are branches. Conceivably, as long as the algorithms are reasonably complex, the chances for different basic blocks to form the same CFG are slim. As an experiment, we built the CFGs for various block cipher algorithms such as DES, MARS, Rijndael, RC6, and found out their CFGs are significantly different. In Figure 2, we investigate the similarity of CFGs in the standard C library of the Alpha compiler. There are 1334 procedures in the library file libc.a, with reasonable size (at least 5 basic blocks). We built the CFGs for all these procedures in which each basic block is abstracted as a node (which in fact increases chances of two CFGs being similar). We run the famous graph isomorphism algorithm by Ullman [11] (we reuse the graph matching library developed by Univ. of Naples [12]) between *any two* graphs. In Figure 2, the results show that only 5% of the comparisons find the two graphs match. If we ignore the CFGs with less than 10 basic blocks, only 0.1% match. Finally, if we ignore the CFGs with less than 15 basic blocks, only 0.05% match. This study shows that each CFG has a distinct fingerprint. Therefore, if the programmer reuses a procedure in the library with 10 or more basic blocks, the reuse is almost doomed to be found out by the attacker due to its distinctive fingerprint. Notice that, this estimation is conservative due to our abstraction of the CFGs that ignores sizes of individual basic blocks; if they are included, the number of matches would decrease further. Even if matches occur, the attacker can still narrow down to a few possible procedures that might be reused.

If only partial CFG can be identified, with subgraph matching algorithms[11][12], we can still largely detect the reuses. It is easy to show that the number of legitimate CFG graphs grows exponentially with the number of basic blocks in the CFG; therefore to hide big reuse code is almost impossible. From the prior discussion, CFG, as a matter of fact, can be regarded as the fingerprint of an algorithm.

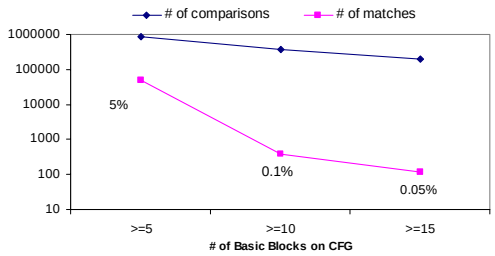

Figure 2. Isomorphic CFG pairs in the standard C library.

Based on the two facts described above, it is quite possible that an attacker can identify the reuse components in a program given its CFG. He can collect the CFGs of all procedures in standard libraries, or for publicly available source code, compile them with a name-brand 3rd party compiler and build the CFGs. By graph matching the program's CFGs with his collection, the attacker can nail down the reuse parts. This not only exposes the reuse code in its entirety, but also helps the attacker in other aspects: 1) A bunch of plaintext/ciphertext pairs for the reuse code are identified. If the hardware cannot afford integrity check due to its prohibitive performance and memory space overhead [4], the attacker might construct a program to read out other

code like in [6]. 2) More critically, although the code is reuse code, all data involved are private to the program. Next, we will show how critical data can be found out in some cases. 3) By watching the interaction between reuse code and the programmer's own code like calling sequence, parameters, the attacker can learn more about programmer's own code.

The technique of CFG matching is merely a particular case of pattern matching, which has been widely used in security applications such as network intrusion detection and computer virus detection.

2.2 Critical Data Leakage via Value-dependent Conditional Branches

CFG matching breaks instruction encryption to some extent. In this section, we further point out that the exposure of instruction access sequence may nullify the data encryption as well.

A typical conditional branch makes a comparison between two values then decides which path to take. Therefore the control flow information, which is exposed completely in the instruction access sequence, can leak important information about the values being compared, although data are encrypted outside the processor and therefore cannot be obtained from the data bus and memory directly.

The following example assumes the algorithm used is known beforehand (most security systems assume the cryptographic algorithms used are known to the attacker) or has been detected by CFG matching. It demonstrates how the critical data (secret key in this case) is revealed.

Example

Diffie-Hellman and RSA private-key operations consist of computing $R = y^x \bmod n$, where the attacker's goal is to find x, the secret key. To show the problem easily, we assume that the implementation uses the simple modular exponentiation algorithm in Figure 3.a, which computes $R = y^x \bmod n$, where x is w bits long. The algorithm is widely used in software implementation, therefore we can reasonably assume the attacker has identified it through CFG matching.

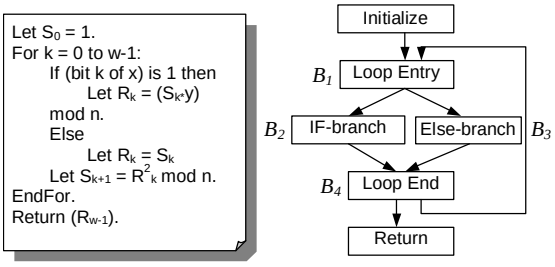

Figure 3. Modular exponentiation algorithm

The corresponding CFG for this small piece of code is shown in Figure 3.b. From Figure 3.a, we can easily find that inside the loop body if the current examined bit of x is 1, IF-branch is executed, otherwise Else-branch is executed. We assume IF-branch resides in address B_2 and Else-branch resides in address B_3 (B_2 and B_3 are different). Since the instructions are not cached, the secure processor must behave as follows: if the current examined bit of x is 1, then fetch the IF-branch in B_2, otherwise, fetch the Else-branch in B_3. This results in a sequence of addresses for B_2 or B_3 showing up on the address bus correspondingly. By monitoring the address bus and capturing the addresses transmitted, the attacker can guess whether the respective bits of x are 0's or 1's and get the secret key x. Even if he cannot distinguish between IF-branch and Else-branch, the information on the address bus leaves only two possible values of x to guess (the correct key or its complement).

This example tells us, if the conditional branch is known to the attacker, the direction of execution path after the branch can expose the outcome of the comparison, which might be helpful in determining or narrowing down the values involved.

Notice that missing several rounds of the for-loop can hide part of the secret key, but still help the attacker substantially to narrow down his search space. It is well known in the security domain, 64-bit encryption has much less strength than 128-bit encryption. If the attacker can capture half of the for loop, his searching space will be cut from $2^{|x|}$ to $2^{|x|/2}$, which is $2^{|x|/2}$ times faster.

This kind of attack is very similar to the timing [15] or power differential attack [16], which has been used to successfully break many commercial smart cards. However, the attack described in our paper is much simpler and more precise since monitoring a high-performance bus activities accurately can be achieved by a simple customized FPGA design as shown in [21].

2.3 Cache does not Help Much

Modern processors now typically consist of large on-chip caches. People might argue that most control flow information is hidden from outside tapping by cache.

However, since cache is a shared resource among all processes running on the processor and as the previous papers, we assume the OS is not secure (please refer to the next section about assumptions). 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 ascertain that his own process fills and occupies most cache space before switching to the process being attacked. In this way, all memory accesses are exposed directly on the address bus due to capacity misses.

Even if only one process is running, many processors have a unified L2 or L3 cache for both code and data. If the program's working set can be affected by inputs, the attacker may intentionally increase the working set size, causing more instruction misses.

Generally, cache is not predictable, especially in multi-process environment. Different parts of the control flow can be leaked during different runs. It is possible that the attacker can finally get the whole picture.

On the other hand, even if the cache can hide some parts of the control flow, information still leaks to some extent. As mentioned before, subgraph matching can match partial CFG. Partial execution path can still be used to prune the searching space for critical data.

Finally, our goal is to completely eliminate the information leakage due to exposed instruction access sequence. Although on-chip cache (or some other techniques as mentioned in the related work section) can somehow confuse the attacker, they do not provide security guarantees.

3. Preliminaries

The Machine Model

In our model, the only trusted hardware is the processor chip, which includes processor core, L1 and L2 cache. To support secure processor requires additional hardware, which can be assumed to be on-die as long as its size is reasonably small. The attacker has full control of the components outside the central chip, which enables him to tap on the bus (even directly inject his own data), and modify the memory contents. This model has been adopted by earlier work [1][2].

Assumptions

In this work, we build our scheme based on the XOM-type architecture[1][2][3][4][5]. In other words, code is encrypted when stored outside the security boundary, prohibiting direct access to the code. In particular, we implement the stream cipher described in [2] as it is very efficient. In addition, we incorporate integrity checking into our scheme to guarantee code cannot be tampered with in memory even after relocation. (5.7) Finally, similar to other prior work, the OS is assumed to be insecure.

Information leaks through data access sequence is possible as well, e.g. stack growth indicating the invocation of a procedure, due to the scope of this paper, we did not explore the opportunities to protect access sequence to the data segments in memory.

System calls can give out control flow information somehow, but the interaction between application and operating system is unavoidable, for example, the result has to be output at last. This problem is actually left to the programmer to not to put system calls at sensitive points of the code.

Execution time cannot be hidden due to its tight association with performance. It is normally unreasonable to require the program to run for the same amount of time regardless of inputs.

We assume code size is not sensitive information, therefore can be exposed to the attacker.

Finally, we assume code misses and data misses to memory are distinguishable. An additional bit can be attached to instruction fetch requests. If a request misses in higher level caches, this bit is carried to the lower level cache request, until the fetch request reaches memory along with this bit.

Objective

We put security at the highest priority, and intend to avoid any information leakage through the instruction access sequence. Although performance degradation is inevitable, it should be properly controlled within a tolerable range for common cases. For the worst cases, security should still be guaranteed.

4. Independent Access Sequence

It is very difficult to quantify how much information is leaked through the instruction access sequence. As mentioned above, even partial information can be exploited by the attacker to ease his hacking. However, the following claim is sufficient to guarantee the attacker cannot learn anything by monitoring the instruction access sequence.

If the instruction access sequence is always independent of program execution, no information will be leaked by monitoring the sequence.

To achieve *independent or fixed* instruction access sequence, a naïve way is to read the whole code segment from the beginning to the end repeatedly. As shown in Figure 4.a, in each round, the whole code segment is read from beginning to the end once. No matter what is going on inside the processor, the processor always reads code blocks in this fixed sequence. Apparently, the only thing exposed is the code size, which has been assumed to be insensitive information. However, the naïve approach can lead to significant slowdown. When there is a miss fetch request from the cache, the request cannot be satisfied immediately from the memory, but has to wait till the block is read in through the fixed sequence. When the code segment is large, the processor may have to stall for a long time. For example, in Figure 4.b, if the current reading block is block 70, and the pending miss fetch requests are block 100, 200 and 50, the processor has to wait for 30 reads to block 100. Again, it has to wait another 100 reads to get block 200. To get block 50, it needs to wait till the access sequence reaches the end of the code segment (finish this round of reading), and starts from the beginning until block 50 is read in. This delay can be enormous if the code segment is very big. Conceivably, the naïve approach will cause intolerable performance loss.

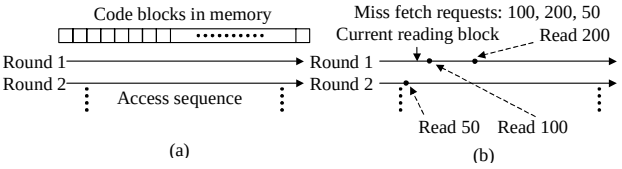

Figure 4. Independent instruction access sequence.

To reduce the stall time, intuitively, we should move code blocks that have a high probability to be requested in the near future to a place that can be quickly referenced. In next section, we will present HIDE, a solution that improves performance without losing the security guarantee.

5. HIDE

5.1 Overview

HIDE: Hardware-support for leakage-Immune Dynamic Execution. We use hardware support to prevent information leakage through the instruction access sequence. Here, "dynamic" means code blocks might be dynamically relocated during the execution.

HIDE dynamically maintains a *hot function set*, which keeps track of the most recently referenced functions at runtime. Based on the hot function set, code blocks belonging to hot functions are prefetched to an on-chip cache or a secondary, fixed, but faster access sequence in memory.

The idea is quite close to the memory hierarchy. We dynamically relocate code blocks that will be most likely used in the near future close to the processor. The two approaches give us a design tradeoff, i.e. on-chip area vs. memory bandwidth. We first define some terms to develop our approach.

5.2 Some Basic Concepts

Access Ring (AR): Represents an abstract structure in the form of a ring which when continually traversed results in a fixed access sequence. AR starts at one address called *Ring Start Address*. After accessing sequentially till the *Ring End Address*, it returns to the Ring Start Address and continues with the same access pattern. In short, it accesses a contiguous piece of memory sequentially and repeatedly.

Read Access Ring (RAR): A special AR in which each access of the AR is a read access.

Substitution Access: a memory operation, which reads from a memory location, then writes back to the same location. The write-back data may or may not be the same as the read-in data. The attacker cannot tell if the data is the same or not, because it is re-encrypted using a different key when written back.

Substitution Access Ring (SAR): each access of the AR is a substitution access.

Fetch Request Queue: a small queue of unsatisfied cache miss fetches is maintained on-chip. Upon read from the ring the blocks are de-queued. This queue is normally small, since the processor will stall if the requested code block cannot be fetched. Only when it speculates on several paths, can the queue have multiple active requests.

AR can be quantified by the following metrics: *Range, Round, Speed, Period and Bandwidth*. *AR Range* is the size of memory region between ring start address and ring end address. Each *Round* is one iteration of accesses from the start address to the end address. *(Turnaround) Speed* is defined as the number of rounds accessed per time unit. *AR Period* is the time one round takes, i.e. the reciprocal of speed. *AR Bandwidth* is the bandwidth the ring takes for memory accesses. Thus, the following holds good: $RAR_{period}=RAR_{range}/RAR_{bandwidth}$ and $RAR_{speed}=RAR_{bandwidth}/RAR_{range}$. For SAR, it is $SAR_{period}= 2*SAR_{range}/RAR_{bandwidth}$ and $SAR_{speed}= RAR_{bandwidth}/(2*SAR_{range})$, since SAR has both read and write operations to an address.

We also have the following claim.

If the memory access bandwidth is fixed, then RAR (or SAR) with larger range must have slower (turnaround) speed.

The naïve approach mentioned in previous section can be regarded as a single monolithic RAR ranging over the entire code segment in memory. This RAR is also called *Base RAR*.

The speed of base RAR can be extremely slow when code size (the range of the RAR) becomes large. Assuming processor fetch requests are uniformly distributed over the entire code segment, the average stall time is $1/(2*speed_{baseRAR})$, or $\frac{1}{2} *period_{baseRAR}$.

5.3 Hot Function Set

The hot function set is a small on-chip array, which dynamically tracks the most recently used functions using LRU replacement algorithm. Each entry is a pair of memory addresses, i.e. the start and end addresses of the function in memory. When a code block is read from the ring, its address is compared in parallel against all entries to find out if it belongs to one of the hot functions. To build up the hot function set, we insert a special instruction at the head of each function. The instruction simply specifies the size of the function. When the processor reads in such instruction, it first checks if the function is already in the set. If so, the function entry is relocated to the front of the array. Otherwise, the function is added to the front of the array and all the other elements are pushed down by one slot. The oldest function is kicked out of the hot function set.

The motivation behind a function level prefetching is that the base RAR is very slow, so prefetching must be done aggressively to reduce the stall time of fetch misses. We feel function calls form the right coarse granularity boundary for such a purpose.

5.4 Prefetch Buffer

Prefetch buffer is an on-chip cache-style buffer to store the prefetched code blocks. Cache misses are first looked up in the prefetch buffer before being sent to the fetch request queue. The prefetch buffer is normally organized like a set-associative cache. Blocks read from the Base RAR are checked with hot function set to decide if they should be prefetched into the prefetch buffer. Figure 5 shows prefetch buffer operations.

Read-in from base RAR

```
If there is matching request in fetch request queue then
   Return the block to cache.
Else If it belongs to hot fun set then
        Put to the prefetch buffer.
     Else
        Discard it.
```

Figure 5. Prefetch buffer operations.

A prefetch buffer takes on-chip space. Inside a processor, there may be many processes concurrently running, so a dedicated prefetch buffer for each process is not realistic. Under our scheme, we assume the prefetch buffer is shared among all the processes. During context switch, the prefetch buffer can be saved as a part of the process state, or we can simply flush it.

5.5 Prefetch Ring

Alternatively, in addition to the base RAR, a secondary ring with smaller range and faster speed can be constructed in memory. Figure 6 shows the prefetch ring architecture. A *Prefetch SAR* is constructed in memory. Typically, the prefetch SAR takes less bandwidth than the base RAR, but its range is much smaller, so its speed is much faster than the base RAR.

On the processor side, the read-in code blocks from both rings are sent to the *Prefetch Controller*. The prefetch controller will lookup in the fetch request queue to see if there is a request for that block and decide if prefetching should be done from the base RAR to the prefetch SAR according to hot function set. We will discuss the prefetch controller in detail later.

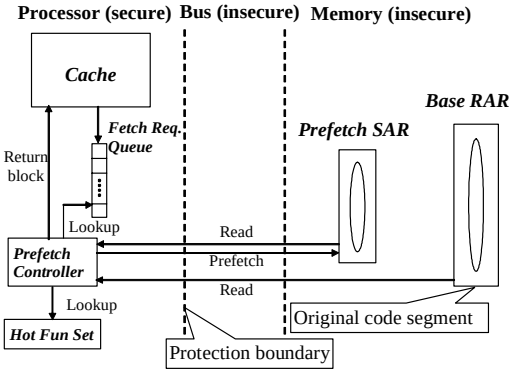

Figure 6. Prefetch Ring.

Base RAR:

start at 0, 10000 blocks, bandwidth 2 unit

Prefetch SAR:

start at 10000, 100 blocks, bandwidth 1 unit

Access Sequence:

RD 0, RD 1, RD 2, RD 3, RD 10000, WR 10000,
RD 4, RD 5, RD 6, RD 7, RD 10001, WR 10001,
…..
…..
RD 396, RD 397, RD 398, RD 399, RD 10099, WR 10099,
RD 400, RD 401, RD 402, RD 403, RD 10000, WR 10000,
…..

Figure 7. Access sequence for the example.

Access Sequence and Code Block Relocation

The two rings can be allocated with different amount of memory bandwidth. We give an example in Figure 7 to show how the access sequence looks like.

Assuming each block takes one unit in the address space. The base RAR starts at 0 and its range is 10000. The prefetch SAR starts at 10000, and the range is 100. Note the prefetch SAR needs both read and write operations. The bandwidth given to Base RAR is twice of that for prefetch SAR.

The memory activities can be grouped into many steps. In each step, it reads four blocks from Base RAR, reads and writes one block from prefetch SAR. Accesses to each ring will loop back to the ring start address when it reaches the ring end address.

Initially, all code blocks in Base RAR are encrypted. All code blocks in SARs are NULL blocks. Notice that NULL blocks are not distinguishable from useful code blocks after encryption. Also two NULL blocks can be entirely different after encryption. We add one additional bit to each block to indicate whether this block is a NULL block. If it is, its content is random.

For the prefetch SAR, code blocks (the read-in ones from prefetch SAR or the ones prefetched from the base RAR) are re-encrypted when writing back. Encryption/Decryption can be made very efficient as shown in [2][3]. Decryption operation at most time costs only one cycle to do XOR under stream cipher. On the other hand, encryption latency is largely hidden by the write buffer. With re-encryption, the attacker cannot correlate any write-out blocks with any read-in blocks, because each encryption is using different keys under stream cipher. Therefore, the processor can relocate code blocks at will, and the attacker has no chance to trace relocated blocks.

Security Strength

Compared to the naïve approach, the access sequence with prefetch SAR is still independent of the program execution. Outside tapping only observes fixed two-ring accesses. Which blocks are prefetched could leak information if no counter measure is taken. In our scheme, whenever a code block is written out, it is encrypted using a different key. So the attacker cannot correlate code blocks and find out which blocks are actually prefetched.

Prefetch Controller

The prefetch controller manages how to prefetch blocks to the prefetch SAR. The selection of blocks to be prefetched is based on the hot function set. But the decision of which block should be kicked out from the prefetch SAR needs explanation. We add an age field to each block in the prefetch SAR. The age field indicates how many rounds a block has stayed in the SAR. Ideally, the oldest block should be kicked out of the ring to accommodate the prefetched block. However, the oldest block might be far away from the current ring access point. Actually, it is not necessary to always pick the “oldest”. Therefore, a

Prefetch SAR Monitor is added, which decides if the current read-in block from the prefetch SAR can be categorized as "relatively old". In our implementation, the monitor gives a cut-off age in each round, so that the number of blocks older than the cut-off age is roughly ¼ of total blocks in the ring. Those blocks are candidates for replacement.

All operations are listed in Figure 9. Note encryption/ decryption operations are performed in parallel with prefetch ring operations to improve efficiency.

Each read-in block from base RAR is checked for matching fetch request. If the block is not requested but belongs to the hot function set, it is put to the output queue, waiting to be prefetched into the prefetch SAR. Read-in blocks from the prefetch SAR are checked for request similarly. If the block is requested or is NULL, a block from the output queue can be written back to the current position of prefetch SAR, otherwise the age of the read-in block is checked to see if it is replaceable. If the read-in block is older than the cut-off age, it is replaced by a block in the output queue, otherwise its age is increased and written back to prefetch SAR. All write backs must be re-encrypted. Finally, Figure 10 shows data structures for the prefetch ring, fetch request queue and the output queue.

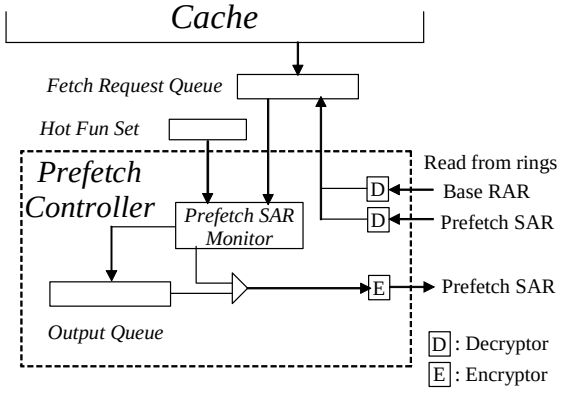

Figure 8. Prefetch controller.

Read-in from base RAR

IF there is a matching request in fetch request queue THEN
  Return the block to cache.
ELSE IF it belongs to hot fun set THEN
  Put to output queue to prefetch SAR.
  ELSE
  Discard it.

Read-in from prefetch SAR

IF there is a matching request in fetch request queue OR the block is NULL THEN
  IF the block is not NULL THEN
  Return the block to cache.
  IF output queue is not empty THEN
  Encrypt one block from output queue and write to prefetch SAR
  ELSE
  Encrypt and write back a NULL block to prefetch SAR
ELSE
  IF block is older than cutoff age AND output queue not empty THEN
  Encrypt one block from output queue and write to prefetch SAR
  ELSE
  Increase age, encrypt and write back the read-in block to prefetch SAR

Figure 9. Prefetch ring operations.

Overhead Analysis

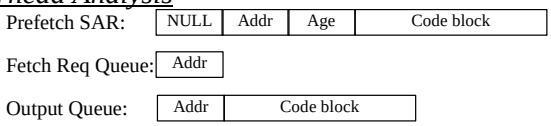

Figure 10. Data structures.

The space taken by the prefetch SAR is far smaller than the original code space, although it needs 3 additional fields. The NULL field is only 1 bit and the age field is set to 4 bits (age saturates at 15 after more than 15 rounds). We use cache block addresses, so the addr field is 32-5=27 bits, assuming 32B block size. Thus, the space overhead due to these fields is 32/256=12.5%. Both prefetch request queue and output queue contain no more than 20 entries.

5.6 Other Approaches

Our hot function set based prefetching algorithm is conservative, since if a function is called first time, there is no way for it to prefetch the code blocks of that function. We also studied call-graph based prefetching algorithm, in which all the possible descendants of hot functions down to a certain level will be prefetched. It is more aggressive but we found its performance is no better or even worse than simple hot function set based prefetching in most cases.

We spent a lot of efforts to dig out the reason behind this. After delving into the code layout generated by GCC, we found that in over 70% cases the callee function is stored in a lower address close to the caller function. This is simply because that C language requires callees defined prior to the caller if no prototype is given, while GCC generates code mostly according to the order in the source file. This kind of function layout generates pathological cases for the call graph based prefetching. After reaching the caller function, the prefetching algorithm will decide to prefetch the possible callees. However, the ring accesses have already gone past callees in the current round of ring accesses (since they appeared before callers in the code layout). Thus, the processor has to wait for another round to prefetch them, which makes prefetching pointless. Currently, we are not able to re-layout program at architectural level. Future work could involve compiler optimizations to help the call-graph based prefetching by laying the functions topologically according to the call graph.

It is also possible to have multiple prefetch SARs in memory that have different range and speed thus build a prefetch ring hierarchy. We can use conservative prefetching algorithm to prefetch blocks into faster and smaller rings, on the other hand aggressive prefetching algorithm to prefetch blocks into bigger and slower rings, in hope of achieving the similar effect of memory hierarchy. However, to make access sequence independent, these rings have to be accessed all the time with fixed pattern, therefore more bandwidth will be consumed. We have experimented with multiple prefetch rings, but performance always degrades if bandwidth is taken from the base RAR. The reason is aforementioned, aggressive prefetching does not help much if the program layout is

inherently poor.

Moreover, there are many other ways to construct independent memory access sequences. For example, we can read in and store a bunch of code blocks in a buffer then write them back to new locations after some time to confuse the attacker (we call this approach as shuffle buffer), or we can choose memory access locations according to a random distribution [7][8]. However, these approaches either cannot guarantee zero information leakage (the former case) or cannot work efficiently in real world (the latter case).

5.7 Other Considerations

Integrity checking is needed to ensure code is not changed in memory. Hash tree [4] can be used for this purpose. Alternatively, we can build the hash tree based on the ring architecture. Each level of the hash tree nodes are put on a *Hash Ring*, with tree root staying on-chip. The ring being checked (at lowest level) and all its hash rings are read in sequentially, so each level of ring data can be checked with their hash values in the higher level ring. For an m-ary hash tree (i.e. m units of data are hash to 1 unit hash value), The range of the k+1 level hash ring is 1/m of the k level hash ring. After reading in m units of data from level k hash ring, we read 1 unit of k+1 level hash for checking. So the bandwidth overhead is $1+\sum_{k=1}^{\infty}1/m^k = 1+1/(m-1)$.

In a multi-process environment, a large prefetch buffer can cause slowdown due to context switches if we choose to store it as a part of process state, as mentioned in section 5.4. On the other hand, the prefetch ring takes very small on-chip space, therefore is more scalable with the number of active programs.

As mentioned before, the attacker may deprive most cache space from the program being attacked. This can lead to extremely severe slowdown for the attacked program due to long waiting time on the ring. However, our scheme can still guarantee no information is leaked under this worst case. Moreover, we believe it actually penalizes the attacker in some sense; a very slowly running/stalled program is likely to frustrate the attacker from trying to snoop to discover "interesting" program behavior.

The benchmarks used in our evaluation are from SPEC2000. Their sizes are not very big. For large-scale applications, the base RAR can be huge thus very slow. So, more bandwidth is required to speedup the access rings. Further, real applications might rely on shared libraries that can be much bigger than the application itself. Maintaining access rings to the shared library will cause bandwidth crisis. It is also not efficient to incorporate the shared library to the rings of every application that invokes it. A better solution could be like in Figure 11. Only the processor chip and the bank controller chips are assumed to be secure. All buses and memory banks are not secure. Program code is split and stored in separate banks in encrypted form. Multiple rings are maintained in separate banks with fixed access sequence. The bus bandwidth between each bank controller and its memory bank can be very high, so the ring speed is high too. The communication channels between the processor and the bank controllers are allocated with low bandwidth. Fetch requests and return blocks are transmitted through these channels in encrypted form (we have to transmit some random data to achieve independent access sequences in the communication channels). Code in shared libraries is put in rings shared by multiple programs. Prefetch buffers can be moved to the bank controllers and prefetch rings might be constructed in memory banks by the bank controllers as well.

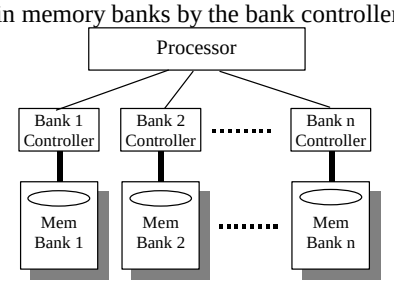

Figure 11. Multiple rings with decoupled memory accesses.

6. Performance Evaluation

Table 1. Default architectural parameters.

Clock frequency	1 GHz	L1 I/D	DM, 8K, 1 cycle 32B block
Fetch queue	32 entries	Unified L2	4way, 32B block 512K (8 cycles) or 1M (12 cycles)
Decode width	8	Memory bus	200M, 8 Byte wide
Issue width	8	Memory latency	first chunk: 80 cycles, inter chunk: 5 cycles
Commit width	8	SNC cache	64K, 4way, 32B
RUU size	128	Encryption/ Decryption latency	50ns
LSQ size	64	Base RAR	400MB/s bandwidth
Branch predictor	Perfect	Prefetch buffer	8K, 4way, 32B
TLB miss	30 cycles	Prefetch SAR range	1/20 of the code size
Hash tree	4-ary	Prefetch SAR BW	200MB/s
		Hot function set	20 entries

We evaluate our schemes on a processor model with default parameters in Table 1. All SPEC2000 integer benchmarks are used as representative applications. Implementation is done with the Simplescalar toolset [17]. We perform our experiments based on SimPoint [20] to capture the characteristics of benchmarks accurately. Each benchmark is fast-forwarded according to SimPoint then simulated by 100M instructions. Memory bus is 8 Byte wide and fully pipelined running at 200MHz. The base RAR is allocated with 400MB/s bandwidth, which is ¼ of the total bandwidth available. Clearly, the accesses to ring are all sequential and can be easily pipelined. We have two configurations for the unified L2 cache—512K or 1M. The encryption/decryption mechanism is built upon [2], incurring very little performance degradation. The SNC cache size and encryption/decryption latency are shown in the table. Hash tree is 4-ary (33% bandwidth overhead). Default prefetch buffer size is 16K and the prefetch SAR range is 1/20 of the total code size, bandwidth is 200MB/s (1/2 of the base RAR).

IPC Comparison with Default Model

Next, we compare the IPC for the two approaches (prefetch buffer and prefetch SAR) with the original IPC and the IPC with only the base RAR in Figure 12 and Figure 13. For clarity, we normalized the IPC numbers to the one with only one base RAR. Absolute IPC values are also shown in the figure. The long horizontal line marks where normalized IPC=1. Small horizontal lines above the bars are the original IPC after normalization. For each benchmark, the left bar is for the prefetch buffer, while the right bar is for the prefetch ring. Figure 12 evaluates the performance under 512K L2 model. We observe both approaches achieve big improvements over the base ring only case. The performance of the two approaches is very close. In half of the benchmarks, prefetch buffer is better, while in the other half, prefetch ring is better. For benchmarks with small degradation like bzip2 and mcf, the improvements of both prefetch schemes are also limited due to reduced optimization space. With only the base ring, the degradation from the original IPC is huge for some benchmarks like gcc (99.7%), vpr (93%), etc. 5 of the benchmarks have their normalized original IPC number out of the range of the figure. On average, base ring only approach degrades the performance by 73%. With prefetch buffer the degradation is reduced to 53%. With prefetch ring, the degradation is 52%. In the meantime, the prefetch buffer gets 71% speedup and the prefetch ring gets 74% speedup over the base ring only case. This is the cost for guaranteed security under our scheme, which is our objective in this work. As we will see later, allocating more prefetch buffer space and more bandwidth can give even bigger improvements. Notice that, the default model chosen by us is not the one with best performance, but the one with relatively small cost in term of chip area or memory bandwidth. With larger prefetch buffer or more bandwidth, we can further reduce the degradation—e.g. as shown later, 64K prefetch buffer can have 23% speedup over the 8K prefetch buffer, which reduce the degradation to about 42%. Perceivably, more costly prefetch buffer or ring can further cut down the degradation.

Figure 13 evaluates the 1M L2 model. The degradation is less than the 512K cache, and not surprisingly, the approaches get less improvement too. The prefetch buffer still performs closely with the prefetch ring. On average, the performance degradation for base ring only, prefetch buffer and prefetch ring are 65%, 45% and 44% respectively. The prefetch buffer improves 57% over the base ring only case, and the prefetch ring improves by 60%.

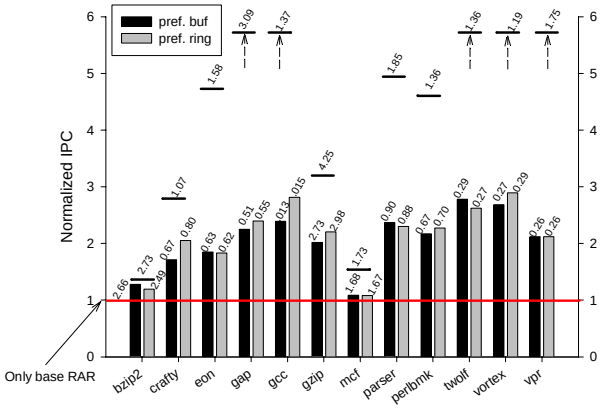

Figure 12. IPC comparison with default model (512K L2).

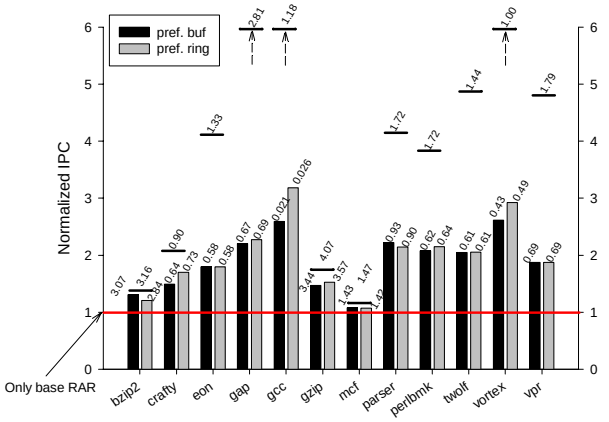

Figure 13. IPC comparison with default model (1M L2).

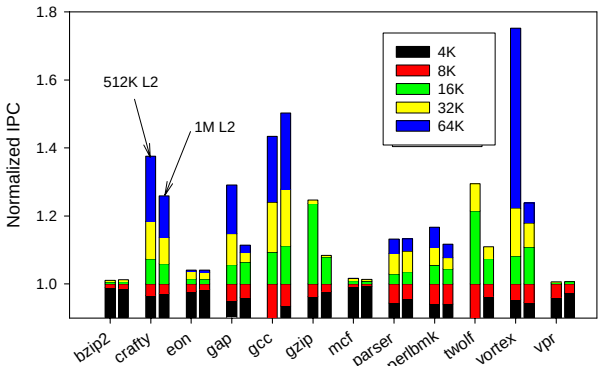

Figure 14. IPC comparison for prefetch buffer sizes.

Prefetch Buffer Size Sensitivity Study

Figure 14 shows how prefetch buffer size affects the performance. We vary the prefetch buffer size based on the default model. For comparison, all IPCs are normalized to the

one with 16K prefetch buffer. For each benchmark, we have one bar for 512K L2 cache and the other bar for 1M L2 cache. From the figure, we observe the IPC number grows monotonously with the prefetch buffer size. For some benchmarks like crafty, gap, gcc, vortex, larger prefetch buffer can clearly increase the IPC, while the others either have little improvement or saturate after either 16K or 32K size buffer. Another observation is the ones with 512K cache benefit more from large prefetch buffer. It is probably because more miss fetches are generated (except for gcc, its IPC is already very low for both cache sizes). On average, for 512K L2, IPC improvements against the default model are –15%(4K), 7%(16K), 13%(32K), 23%(64K). For 1M L2, the improvements are –4% (4K), 5%(16K), 9%(32K), 13%(64K). With a 64K prefetch buffer, the performance degradation is 42% for 512K cache and 38% for 1M L2 cache.

Prefetch Ring Sensitivity Study

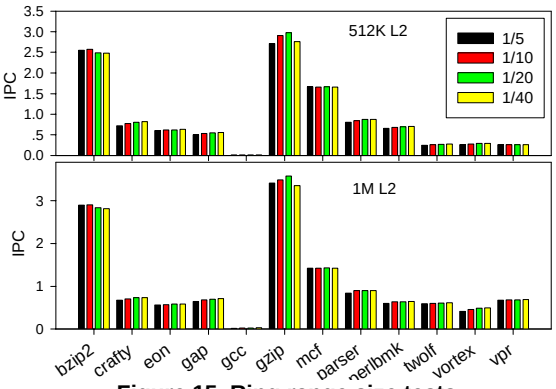

Figure 15. Ring range size tests.

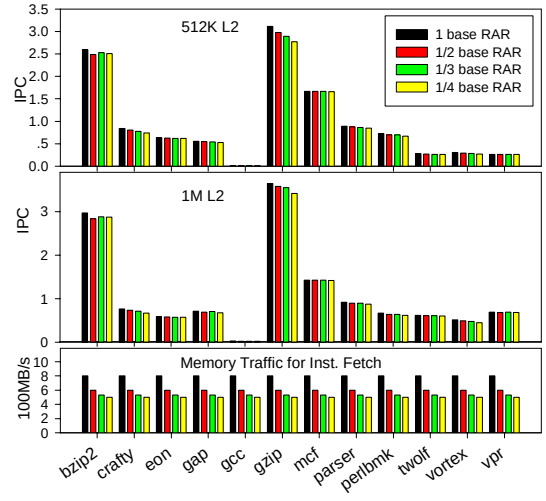

Figure 16. Ring bandwidth study.

Next, we look into parameters of the prefetch ring. In Figure 15, we evaluate how the ring range affects performance. Four specific cases are studied. The prefetch ring range is set to 1/5, 1/10, 1/20 and 1/40 of the original code space respectively. For both cache sizes, most benchmarks can gain a little speedup when ring range becomes smaller, then slightly decrease or saturate when ring is too small. As shown in section 5.2, with fixed ring bandwidth, small range ring has faster speed. However, the number of code blocks that can be prefetched and stored on small range ring is also much fewer than the big range ring. When the prefetch ring becomes too small, some prefetched blocks cannot stay till they are referenced, which could lead to performance loss. Especially for gzip, bzip2 and mcf, whose original code sizes are already very small, the IPC decreases more when the prefetch ring is squeezed.

On average, for 512K cache, the IPC improvements over the 1/5 ring are 3.2%(1/10), 5.0%(1/20), 2.8%(1/40), and for 1M L2, the improvements are 3.1%(1/10), 4.9%(1/20), 2.8%(1/40). Thus, the ring range affects the performance less than the prefetch buffer size, and it is also very insensitive to the cache size. This is probably because prefetch ring can contain much more code blocks than the small prefetch buffer in most cases.

In Figure 16, another parameter--the ring bandwidth is varied, while keeping all other default configurations. The base RAR is running at the same speed as in the default model. We allocate 4 different bandwidth to the prefetch ring, i.e. ¼ of the base RAR, 1/3 of the base RAR, ½ of the base RAR and the same as the base RAR. At the bottom of Figure 16, we show the memory traffic accordingly. In most cases, more bandwidth leads to higher IPC. The reason is that most misses can be resolved earlier with a faster ring (as shown in section 5.2, for fixed ring range, more bandwidth leads to a faster ring). This is more significant when cache is 512K and IPC is lower.

In our experiments, we also attempted to lower the bandwidth of the base RAR and give the prefetch SAR more bandwidth. However, the performance actually suffers. The reason is that due to the poor code layout, the performance of prefetching algorithm is limited. There will be a considerable amount of cache misses that miss in prefetch ring again and turn to the base ring. Reducing the bandwidth of the base ring can result in performance loss due to those prefetch ring misses.

Hot Function Set Size Sensitivity Study

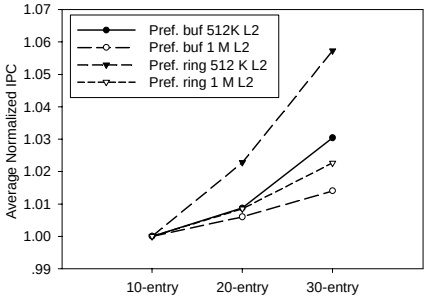

Figure 17. IPC comparison with different sizes of the hot function set.

Next, we evaluate how the hot function set size influences the IPC. In Figure 17, we first normalize the IPCs to the IPC under 10 entry hot function set. After normalization, big IPC benchmarks get the same representational weight as small IPC benchmarks. Then we take the average of all normalized IPCs. The average number is presented as the Y-axis value in the figure. Figure 17 shows the trend how IPC grows with bigger hot function set. It is worth noticing that for 1M L2, the increase is much less than the 512K L2. Also, prefetch ring benefits more from a big hot function set than the prefetch buffer. The first phenomenon can be explained as follows: with a small cache, more functions cannot be entirely hidden inside the cache, thus, a big function set is more desirable to trace those functions. The second phenomenon can be explained as: the prefetch ring is much bigger than the prefetch buffer. For the prefetch buffer, even if more hot functions are traced, it probably cannot hold them.

The overall improvements due to a larger hot function set are limited. From the figure, the maximal speed up is 6%.

Combined Prefetch Buffer and Ring

Prefetch buffer and ring can be combined to further improve the performance. However, a combined approach has many design parameters, like prefetch buffer size, ring speed, ring bandwidth, etc. To better explore the design space, we use a 3D figure in Figure 18. The vertical axis is the average normalized IPC number, we first normalize all IPCs to the IPC with default configuration. Then all normalized IPCs are averaged to become one point in the figure. The other two dimensions are for prefetch buffer (X axis) and prefetch ring (Y axis) respectively. On the Y axis, 4 types of ring ranges and 4 types of ring bandwidth form 16 combinations, e.g. 1/40R-1/2B means that ring range is 1/40 of the original code and ring bandwidth is ½ of the base RAR.

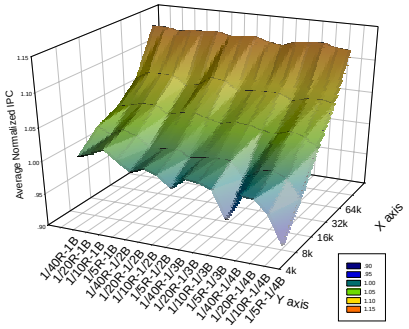

Figure 18. Design space for combined buffer & ring(512K L2)

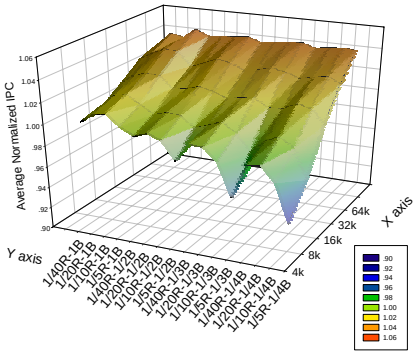

Figure 19. Design space for combined buffer & ring(1M L2)

From Figure 18, it is not surprising that the combined approach can achieve better performance. The best case can reduce performance degradation to 38%. However, the incremental improvements are marginal considering the hardware costs for both ring and buffer. Conceivably, prefetch buffer and ring may have many duplicated code blocks, wasting bandwidth and area. Clearly, the IPC can be improved most significantly with larger prefetch buffer. Given a fixed prefetch buffer size, we see waves along the Y-axis. Each wave has the same ring bandwidth. With more bandwidth the waves go higher. Inside each wave, as ring range decreases, the IPC first increases then decreases as explained earlier.

The figure gives architects many choices to implement the combined scheme. If the on-chip area is not a concern, large prefetch buffer is a good choice. With big prefetch buffer, waves become flat, because the program working set can be mostly fit in the buffer. The architect can pick a relatively small and low bandwidth ring. If on-chip space is limited, but memory bandwidth is relatively cheap, giving the prefetch ring more bandwidth is quite effective. Finally, if both prefetch buffer size and ring bandwidth are set, choosing a proper ring range is important, esp. when prefetch buffer is small.

Figure 19 shows the design space for 1M L2. We notice the whole surface is more flat, indicating less improvements due to reduced cache misses. Other properties are very close to the one with 512K cache. The best case can reduce performance degradation to 34%.

7. Related Work

XOM Architecture

The *eXecute Only Memory (XOM)* Architecture [1] was proposed to support copy and tamper resistant software. Both code and data are encrypted outside the security boundary. Recent advances can preserve privacy and integrity for off-chip data and code with reasonable slowdown [references]. However, XOM like architecture fails to address the information leakage through instruction access sequence, which is the main focus of our work. XOM like architecture forms a part of the infrastructure of our security system.

Oblivious RAM

Goldreich [7][8] realized that software protection should prevent the attacker from "learning the program" by experimenting with it, e.g. feeding it with different inputs and monitoring the different behavior of the secure processor. In this work, they studied a specific problem of hiding program memory access pattern, which is a major step towards a leakage-protection processor and is also the problem we try to solve in this paper. Their work studied the problem in a theoretical view and proved that the lower bound of running a program without leaking memory access pattern is $t \log_2 m$, in which t is the original running time of the program, m is the number of locations in the external memory. This model is no longer valid in modern processors. One obvious problem is that there is no cache or memory hierarchy in this model, thus every instruction fetch has to go to external memory. Also, architectural issues regarding building a real processor remained unaddressed in their work that we have addressed here investigating different tradeoffs involved in the solution space.

DS5002FP

DS5002FP microcontroller [9] is a widely used commercial bus-encryption processor. This chip not only encrypts data in the memory, but also addresses. Instructions and data are not stored in their original order, but rather in seemingly random locations in memory. To access a code block, the original address must be encoded and the address after encoding is used to access the memory. They claim that due to the bus encryption, it is virtually impossible to determine the original control flow of the program. However, the code locations are never changed in memory during program execution. In other words, an original address is always translated to the same address that is sent to the memory. For example, if a loop contains blocks: 100,101,102, and block 100 is actually at address 231 (so 100 will be encoded as 231), block 101 is at 371, block 102 is at 483. Although after encoding, the loop is not stored in contiguous memory locations, the attacker can still detect the sequence 231-371-483 to appear repeatedly as the loop executes. Branches cannot be hidden either. If 102 (483) has two successors: 103 (876) and 104 (751), then by observing two access sequences 483-876 and 483-751, the attacker can conclude there is a branch after block 483. The sequences before and after encryption have exactly the same nature. Therefore, simply permutating code blocks in memory through static encryption does not help at all. On the other hand, it is impossible to encode the same address differently each time, since the code block is at fixed location in memory.

DS5002FP also tries to confuse the attacker by issuing random read accesses between actual read accesses generated by the program. However, this does not work either. Assuming the program enters a loop that runs for a large number of iterations. It quickly turns out, among the access addresses on the instruction address bus, a number of addresses (those actually in the loop) always repeat, while others (randomly generated ones) only appear occasionally. After the actual access addresses are identified, the attacker can simply ignore other random reads.

DS5002FP has been actually completely cracked by Kuhn [6]. The paper describes how the author found out the correspondence between instructions and their ciphertext. Using this, he could construct a program to read out the secret key.

Code Obfuscation

Code obfuscation has been proposed to change the code structures at compilation time to make it very different from the original code. The main goal of code obfuscation is to stop reverse engineering a program using dis-assemblers, debuggers etc. While static control is made difficult to crack, dynamic control flow if monitored could still reveal critical program information. Thus, code obfuscation cannot provide security guarantee [10] but only makes it relatively hard to crack the code. In our work we are providing security guarantees although the performance is somewhat degraded. We also propose mechanisms to reduce the overheads arising out of these guarantees.

8. Conclusion

Providing security guarantees to the problem of software rights protection imply not leaking any control flow information which could generate a unique fingerprint that could be matched compromising intellectual property of the software. In this work, we devise processor mechanisms that establish such guarantees. However, establishing such guarantees entails performance loss. We propose different mechanisms involving prefetch ring, prefetch buffer to alleviate such a loss. We then provide a design space exploration that measures the impact of different parameters and proposes solutions under different constraints (such as the on-chip area etc.). The final recommendations are: with enough on-chip space, prefetch buffer provides better performance in general, otherwise if bandwidth is cheap, prefetch ring is more suitable. When both of them are cheap, we can use a combined approach to further improve performance.

REFERENCES

[1] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," *ASPLOSIX*, Nov. 2000.

[2] J.Yang, Y.Zhang, L.Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering," *In Proc. 36th International Symposium on Microarchitecture, to appear*, Dec. 2003.

[3] E.Suh, D.Clarke, B.Gassend, M.v.Dijk, S.Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors", *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, December 2003.

[4] B.Gassend, G.E.Suh, D.Clarke, M.v.Dijk, S.Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification", *The 9th International Symposium on High Performance Computer Architecture (HPCA9)*,Feb. 2003.

[5] G. E.Suh, D.Clarke, B.Gassend, M.v.Dijk, S.Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," *Proceedings of the 17th International Conference on*

Supercomputing, Jun. 2003.

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

[7] 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.

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

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

[10] B.Barak, O.Goldreich, R.Impagliazzo, S.Rudich, A.Sahai, S.Vadhan, K.Yang, "On the (Im) possibility of Obfuscating Program," *CRYPTO 2001*.

[11] J.R.Ullman, "An Algorithm for subgraph Isomorphism," *Journal of ACM*, Vol.23, pp.31-42, 1976.

[12] VFLib Graph Matching Library, http://amalfi.dis.unina.it/graph/db/vflib-2.0/doc/vflib-1.html

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

[14] D. Lie, J. Mitchell, C. A. Thekkath, and M. Horwitz, "Speci-fying and Verifying Hardware for Tamper-Resistant Software," *IEEE Symposium on Security and Privacy*, 2003.

[15] P.C. Kocher, "Timing attacks on implementations of Die-Hellman, RSA, DSS, and other systems," *Proceedings Crypto'96, LNCS 1109*, Springer-Verlag 1996, 104-113.

[16] P. Kocher, J. Jaffe and B. Jun, "Differential Power Analysis", *Proceedings of CRYPTO '99.*

[17] Doug Burger and Todd M. Austin. "The SimpleScalar Tool Set Version 2.0," Technical Report 1342, University of Wisconsin--Madison, May 1997.

[18] Ross Anderson, Markus Kuhn, "Low Cost Attacks on Tamper Resistant Devices," *Proceedings of the 1997 Security Protocols Workshop*, Paris, April 7--9, 1997.

[19] J. Kelsey, B. Schneier, D.Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," *ESORICS '98.*

[20] Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder, "Automatically Characterizing Large Scale Program Behavior," *ASPLOS 2002*, October 2002.

[21] Andrew Huang, "Keeping Secrets in Hardware: the Microsoft Xbox(TM) Case Study," *MIT Artificial Intelligence Laboratory Technical Report AIM-2002-008*, May 26, 2002.