Building Intrusion-Tolerant Secure Software

Tao Zhang Xiaotong Zhuang Santosh Pande
College of Computing
Georgia Institute of Technology
Atlanta, GA, 30332-0280
{zhangtao, xt2000, santosh}@cc.gatech.edu

Abstract

In this work, we develop a secret sharing based compiler solution to achieve confidentiality, integrity and availability (intrusion tolerance) of critical data together, rather than tackling them one by one as in previous approaches. Under our scheme, some critical data values are automatically identified by the compiler, whereas some others are specified by the user. The compiler generates code for scattering/assembling and verifying of those critical data values using secret sharing scheme. In this way, we achieve data confidentiality and integrity. We also provide mechanisms to gracefully recover upon data tampering, achieving intrusion tolerance. The implementation of our secret sharing scheme is carefully crafted to achieve low overhead. We further propose several compiler optimizations such as secret-sharing-aware register allocation, rematerialization etc. to reduce the cost of secret sharing further, making our scheme a practical solution in a high performance system.

1. Introduction

Computer systems are critical for our society to function properly. Numerous mini-computers form parts of critical systems in automobiles, aircrafts, appliances and medical devices to control their operations. At the same time, huge computer systems are integral parts to provide critical facilities such as financial services, telecommunication, transportation, and energy production and distribution networks. The increasing of the society's dependence on computer systems brings great interest (from criminals and malicious users) to break in and tamper those systems. The tampering of computer systems is at best inconvenient; in certain areas, tampering may lead to great chaos, huge financial losses or even loss of human lives. Thus, it is extremely important to secure critical computer systems.

In this work, we focus on protecting data of secure software by a compiler solution with minimum hardware support. Although secure processor research is very active recently [11][12][13][14], the proposed schemes introduce too much modification to the regular processor and will not be available in the near future. So it is still critical to enforce security through software-based solutions, with as little hardware modification as possible to make it practical.

Security implies enforcing several important properties including confidentiality, integrity and availability (intrusion tolerance). Confidentiality prevents the attacker from reading sensitive data stored in software such as passwords, keys, credit card numbers, sensitive personal information etc. The stealing of sensitive data could be very damaging. As an example, according to a recent survey released by Federal Trade Commission [2], 27.3 million Americans have been victims of identity theft in the last five years, including 9.91 million people or 4.6% of the population in year 2002 alone. In 2002, identity theft losses to businesses and financial institutions totaled $47.6 billion and consumer victims reported $5 billion in out-of-pocket expenses.

Integrity provides means to detect tampering to critical software data. Just as sensitive data is vulnerable to theft, critical values in a program such as stack return addresses, jump tables involved in case statements are in fact vulnerable from tampering. These critical values decide the control flow of the program and are the most common starting points of attacks. For example, consider the most common and popular attack involving buffer overflows [1]. The attacker sends a long string as a legal input to the program. Due to the absence of bounds checking, the array overflows and corrupts return location in the stack for a procedure call. The procedure therefore returns to a different location where the attacker has slipped in malignant code that then takes over the control of execution. Another typical scenario involves manipulating the jump table of a case statement to get around access control. Of course, besides critical data that decides control flow, there is other critical data that requires integrity checking such as critical fields in a financial transaction.

Finally, intrusion tolerance recovers data and continues correct software execution even when tampering occurs. Unfortunately, although failure recovery (fault/intrusion tolerance) is often the most important aspect of security engineering, it has not been carefully addressed. Most of the computer security research has dealt with confidentiality and most of the

rest with authenticity and integrity. However, the actual expenditure of systems providing critical services may go the other way around [4]. In this work, we develop a secret sharing based compiler solution to protect software critical data with confidentiality, integrity and availability at the same time, which is never done before.

Introducing intrusion tolerance into secure software is one of the major contributions of our scheme. Intrusion tolerance is critical since no software system can be made perfect. There are always software bugs, configurations errors etc. to be exploited by the attacker. Upon detection of an attack, the simplest response is fail-stop, i.e., shutting down the system to avoid further damage. However, shutting down the system completely leads to inconvenience for the users. For systems providing critical services, it may cause huge financial losses. In areas such as aircraft control, battlefield control, it could be disastrous.

The rest of the paper is organized as follows: Section 2 gives machine model and motivation regarding why we choose this approach; section 3 discusses the overview and implementation details of our secret sharing scheme; section 4 analyzes the security strength of our scheme; section 5 discusses optimizations to reduce the performance overhead introduced by secret sharing operations; section 6 presents results and comparison of secure and non-secure schemes in terms of performance and also effects of the optimizations; section 7 discusses related work and finally section 8 concludes the paper.

2. Background, motivation and assumptions

2.1 Secret sharing

Secret sharing was introduced by Shamir in his famous paper [3] as a way of ensuring secure information exchange that does not rely on encryption/decryption. Informally, a secret (value) is shared by breaking it down into pieces (called shares) that are then distributed. Assume there are *n* shares in total, Shamir's secret sharing scheme has two important properties: 1) knowledge of any *k* or more shares makes the secret easily computable; 2) knowledge of any *k*-1 or fewer shares leaves the secret completely undeterminable, (in other words, it is impossible to guess original value by simply looking at the *k*-1 or fewer shares). Thus, the security value of the scheme relates to the fact that as long as the attacker is not able to gain simultaneous access to *k* number of shares, he has zero information of the secret. In the scheme, *n* and *k* can be chosen freely to get the desired total shares and the threshold to extract the secret.

Formally, Shamir's secret sharing scheme is based on polynomial interpolation: given *k* points in a two dimension plane (x_1, y_1), … (x_k, y_k), with distinct x_i's, there is one and only one polynomial q(x) of degree (*k*-1) such that $q(x_i) = y_i$ for all *i*. The (*k*-1)-degree polynomial $q(x) = a_0 + a_1x + ... + a_{k-1} x^{k-1}$ over the finite field GF(p) (p is a large prime number) is constructed such that the coefficient a_0 is the secret and all other coefficients are random elements in the field. Each of the n shares is a point (x_i, y_i) on the curve defined by the polynomial, where x_i not equal to 0. As an example, choose p as 2147483647 and define field GF(2147483647). Let *k*=3, *n*=5, the secret is 123456789, which could be someone's SSN. One sample polynomial generated is $q(x) = 123456789 + 343194266x + 1300552763x^2$. Let x = 1, 2, 3, 4, 5 then evaluate the polynomial to get 5 shares. The 5 shares generated are 1767203818, 1717089079, 2120596219, 830241591, and 2140992489 respectively.

The core property of Shamir's scheme is that given any *k* shares, the polynomial is uniquely determined and hence the secret a_0 can be computed. However, given *k* - 1 or fewer shares, the secret can be any element in the field.

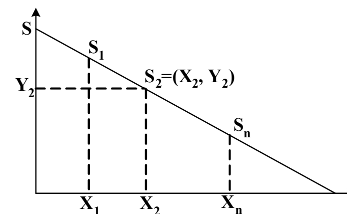

Figure 1. Shamir's secret sharing scheme.

To illustrate the properties of Shamir's scheme, a special case where *k* = 2 (i.e., two shares are required for retrieval of the secret) is given in Figure 1. The polynomial is a line and the secret is the point where the line intersects with the *y*-axis. Each share is a point on the line. Any two shares (two points) determine the line and hence the secret. With just a single share (point), the line can be any line that passes the point, and hence the secret can be any point on the y-axis.

In Shamir's scheme, the higher the degree of the polynomial, the higher the number of shares required to recover the original secret value. Precisely, if the degree of the polynomial is *k*-1, at least *k* shares are needed to reconstruct the original value where $k <= n$. The higher the value of *k*, intuitively it is more difficult to break the security but then one must pay for more computation overheads due to a higher *k*.

There are four types of computation overheads in Shamir's scheme. To get a secret value, *k* shares have to be gathered and then polynomial interpolation based on these *k* shares has to be done. We call the cost to retrieve a secret value as <u>*assembling cost*</u>. When a secret value is to be updated, a new set of *n* shares has to be generated and distributed. We call the overhead as <u>*scattering cost*</u>. There has to be a way to detect the tampering to the shares from the attacker, we call the cost as <u>*verifying cost*</u>.

A simple approach to detect tampering would involve generating secret values using all possible combinations of k shares from all n shares and checking whether values generated are all same. The cost of this approach is C_n^k times of assembling cost. However, a more efficient approach requires only $\lceil n/k \rceil$ values be generated and compared. It is easy to see with $\lceil n/k \rceil$ k-element groups, all n shares can be covered, i.e., used at least once in the secret value computation. As long as we cover all n shares in computation, it is very hard for the attacker to tamper less than n shares of a secret value without being detected. If the attacker is able to tamper all n shares, there is no way to detect that. Besides data confidentiality and integrity protection, secret sharing scheme has the ability to recover tampered data values. Under secret sharing scheme, we can make C_n^k sets of shares, each set containing a particular combination of k shares selected from the n shares. We can assemble a secret value using polynomial interpolation from each set and get C_n^k assembled values. From the assembled values, we take the value that is given out by the majority as the original value. We call the cost recovering cost, which is approximately C_n^k times of assembling cost.

2.2 Secret-sharing vs. traditional approaches

A natural question here is that why we go for secret sharing scheme instead of using traditional techniques. We reason about this below.

The traditional technique to protect data confidentiality is encryption. The traditional technique to ensure data integrity is through checksum or hashing based schemes. Although availability is less studied in security area, it is an important topic in fault tolerant computing. The traditional approaches used there are replication and check-pointing/rollback.

First, it is easy to notice that although confidentiality, integrity and availability are integral properties of security, previous approaches tend to tackle them separately. It is possible to achieve all properties by combining solutions for each. However, that brings great complexity and management issues into the system. It hurts maintainability of the system as well, which is a critical element to achieve system dependability. On the other hand, under our secret sharing scheme, confidentiality, integrity and availability are solved under the same scheme, leading to much better maintainability.

Second, although security research has provided mature solutions to confidentiality and integrity, little research has been done on availability. Traditional solutions to provide availability in fault tolerance area are replication and checking-pointing/roll back. Those solutions cannot be simply copied to security area. Replication is a direct way to provide availability, but it brings complication to confidentiality since if any one of the shallow systems is broken, confidentiality is broken. Moreover, the system cost with n-replication is order of n times larger. Checkpointing requires backing up the whole program state, the size of which could be huge. The resulting overhead prevents fine-grained check pointing and may break real-time constraints of critical real-time software, such as aircraft control, battlefield control etc. For example, in [18], the checkpointing interval is 30 minutes and each check pointing costs several minutes with check pointing size up to tens of megabytes. Also, when rolling back, the whole program state is recovered, which is inefficient since in many cases only a few data values are tampered. On the other hand, our secret sharing scheme is able to recover a tampered data value much faster and only recovers tampered values. We will show in the results section (Table 1) that the computational cost of recovering function is in terms of hundreds of cycles. Considering other overheads of exception handling in which the value actually gets recovered (details presented in section 3.4), our scheme should be able to recover a value within several milliseconds, which is magnitudes smaller than the cost of checkpointing. In summary, our secret sharing recovering scheme could be deployed when fast recovery is desired such as for all kinds of real-time software. On the other hand, our scheme can only recover tampered values successfully when the tampering is limited and there are enough intact shares. Checkpointing/replication has to be deployed if we want to recover from denial of service attacks in which the attacker could erase the whole memory content thus no secret shares can be preserved.

2.3 Machine model and assumptions

We observe that processors without any security support, such as Pentium, are dominating the market currently and will continue to do so in the near future. Any scheme introducing too much modification to current general processor architecture is impractical and given long processor development cycles will not happen at least in near future. On the other hand, a pure software scheme cannot be secure at all. Thus, in our machine model, we only introduce minimum hardware support for security. We assume our target processor possesses a secret private key and is secure. There is a trusted secure kernel residing on the processor, which is implemented in software and provides very basic services for context switching, TLB misses handling, page faults handling and encryption/decryption etc. Other software (OS included) is not trusted. Each process is associated with a secret key and process contexts (including registers) are encrypted using the key during context switches (thus are protected). Minimum hardware additions to a general processor are

assumed, including process key tables, several special registers etc., but without any special crypto hardware.

We have the following security assumptions in our machine model: 1) any hardware component inside the processor is considered to be secure; 2) process context is considered to be secure during context switching, including all registers used by the process; 3) anything else other than the processor itself and the process context is considered non-secure, including the external memory. From the above assumptions, any data residing in the external memory is considered non-secure and has to be protected against attacks. On the other hand, data residing in registers and other on-chip storage is considered to be secure.

We assume only part of the data is security sensitive. For instance, stack return addresses, sensitive or critical fields of personal records etc. Other data, such as home addresses, home telephone numbers etc. is not sensitive and may be already publicly available. Application specific sensitive data can be identified by the user. Only sensitive data needs to be protected. According to its sensitivity, protections with different strengths could be applied.

Furthermore, we assume the attacker does not want to simply bring down the system. Instead, he wants to exploit the security holes in the system and obtains benefits illegally. For example, he does not want to kill an ongoing financial transaction. Instead, he wants to modify the critical fields of the transaction without being caught. Thus under our scheme certain attacks are not preventable. For example, if the attacker launches a DOS attack, wiping off the whole memory contents, our scheme cannot recover from it without additional help.

Finally, in this work, we focus on critical data protection. Program code may be optionally protected by the user through other methods. The critical point is that the security of our scheme does not depend on code protection. Our secret sharing implementation is assumed to be known by the attacker. The security of our scheme roots in the secret to drive the secret sharing algorithm. Detailed security analysis of our scheme is presented in section 4.

3. Our approach

3.1 Overall scheme

Figure 2 shows the phases of our approach. First, the user has the flexibility to specify application specific sensitive data by a simple language extension. In our scheme, we introduce a new variable qualifier "sensitive". The compiler will apply secret sharing scheme to all variables marked by "sensitive" qualifier. Our scheme also identifies application independent sensitive data automatically through compiler analyses. The variables detected include stack return addresses, case jump tables, virtual function tables and any branch targets held in memory.

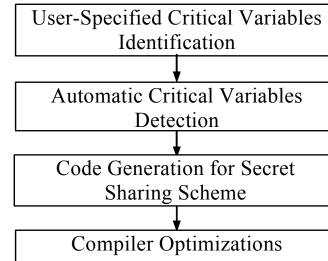

Figure 2. Phases of our secret sharing scheme.

Once all the sensitive data values are identified, the third phase of our approach generates basic secret sharing implementation on all the identified variables by using the threshold k and the total number of shares n determined by the user. The selection of k and n is based on the security requirement of his application. As discussed earlier, the higher k, the higher the number of shares required by the attacker to gain access to the secret value thus the higher confidentiality. On the other hand, the higher n, the higher probability that a tampering to the data value will be detected and the higher probability that the tampering will be successfully recovered. Using the selected k and n, the compiler generates code for the secret sharing scheme, as elaborated in the next section.

Finally, various compiler optimizations are done to reduce the performance overhead due to secret sharing. Optimizations include secret-sharing-aware register allocation, rematerialization, optimizations and versioning of secret sharing functions etc.

3.2 Secret sharing functions

There are four functions implemented to perform secret sharing: scatter(), assemble(), verify() and recover(). Their functionality and pseudo-code are described as follows. The address space for shares is pre-allocated as a huge buffer in heap.

The *scatter*(*addr*, *v, n*, *k*) function takes the address of a value v and scatters v into n shares based on a random polynomial of degree k-1. The actual implementation of scattering function proceeds as follows. First, the original address of v is *XORed with the process key* (described in section 2.3) and fed into a pseudo-random function as seed, then the first n outputs of the pseudo random function are chosen as offsets to the base address of the large buffer allocated, thus the virtual addresses for the n shares are obtained. Then the scattering function randomly generates a degree k-1 polynomial q(x) and calculates n shares as illustrated in section 2.1. The generated n shares are stored into the n virtual addresses

aforementioned respectively. The compiler can simply store to the virtual addresses, with the virtual memory manager taking care of the actual physical memory allocation. Finally, after all the shares are properly updated, the scattering function invalidates the original value v (typically by initializing it to a value which violates the underlying type such as an illegal SSN number, an illegal return address etc.). The pseudo-code for scattering function is shown in Figure 3. Notice that x_i, $i \in [1,n]$ are predetermined values. As a result, we can precompute x_i^1, $x_i^2 \ldots x_i^{k-1}$ for all $i \in [1,n]$. The scattering function only needs to generate random coefficients for $q(x)$ and multiply them with the powers of x_i to get the shares.

```
Input: addr, v, n, k
Output: n shares of v
Algorithm scatter (addr, v, n, k):
Feed addr ⊕ process_key as seed into a pseudo random function f(s),
generate sequence addr1, addr2, … addrn;
Generate a random degree k-1 polynomial q(x), with a0=v;
Evaluate si = q(xi), for i ∈ [1,n];
Store si to addri, for i ∈ [1,n];
```

Figure 3. Pseudo code for scattering function.

The $assemble(addr,n,k)$ function gets the original value by assembling shares generated by the scattering function. It XORs the original address of the value v with the process key to get the seed, feeds the seed into the pseudo random function, then recovers the addresses of the shares belonging to this secret value. After collecting all n addresses of shares, it randomly chooses k addresses, fetches the shares stored in the addresses, then performs polynomial interpolation to recover the secret value v. Although k addresses are enough for recovering the value, we get all n addresses since they will be used in verifying function later. The pseudo code for the assembling function is shown in Figure 4.

```
Input: addr, n, k
Output: v
Algorithm assemble (addr, n, k):
Feed addr ⊕ process_key as seed into the pseudo random function f(s),
generate addresses addr1, addr2, … addrn;
Randomly choose k addresses addr1', addr2', …, addrk'
Fetch si' from addri', for i ∈ [1,k];
Perform polynomial interpolation on s1', s2', …, sk' to get secret value v
```

Figure 4. Pseudo code for assembling function.

The formula for polynomial interpolation is shown in Figure 5. $p(x) = s_i'$ when $x = x_i'$ for $i \in [1,k]$ and $v=p(0)$ as illustrated in Figure 1. The computation is actually simple, since for each term the coefficient of s_i' is determined when x=0. In other words, we can pre-compute those coefficients as constants, so that polynomial interpolation only needs k multiplications and k-1 plus operations.

The $verify(addr,n,k)$ function checks whether the value v generated by the assembling function is valid. The verifying function first constructs $\lceil n/k \rceil$ sets of shares, so that all n shares are covered during the computation with $\lceil n/k \rceil$ sets. Among the sets constructed, one is the same as the set used in assembling function to save some computation overhead. Each set of shares produces a secret value, if any of them is different from another, a tampering is detected and an exception is triggered (description of the exception handler follows). If all the assembled values match, the verifying function finishes successfully. We omit the pseudo code here, since the algorithm is similar to the assembling function except that we run polynomial interpolation for each of the $\lceil n/k \rceil$ sets of shares and compare the assembled values afterwards.

$$
\begin{aligned}
p(x) = {} & \frac{(x-x_2')(x-x_3')\ldots(x-x_k')}{(x_1'-x_2')(x_1'-x_3')\ldots(x_1'-x_k')} s_1' \\
& + \frac{(x-x_1')(x-x_3')\ldots(x-x_k')}{(x_2'-x_1')(x_2'-x_3')\ldots(x_2'-x_k')} s_2' + \ldots \\
& + \frac{(x-x_1')(x-x_2')\ldots(x-x_{k-1}')}{(x_k'-x_1')(x_k'-x_2')\ldots(x_k'-x_{k-1}')} s_k'
\end{aligned}
$$

Figure 5. Polynomial interpolation.

The $recover(addr,n,k)$ function recovers the tampered value when a tampering is detected. The recovering function constructs all possible C_n^k sets of shares, each set containing a particular combination of k shares selected from all n shares. It then assembles a secret value from each set and gets C_n^k assembled values. From the assembled values, we take the value that is given out by the majority as the original value. The recovering function is similar to the assembling and verifying functions except that it assembles all possible sets of shares.

3.3 Share addresses generation

Share addresses generation serves all secret sharing functions. The following requirements must be satisfied when choosing a proper pseudo random number generator to generate share addresses. 1) The scattering function uses the transformed address of the secret value as seed to generate all share addresses, while other functions should be able to regenerate the share addresses using the same original address. 2) Given two different seeds, the output sequences of the pseudo random generator should have a very low collision probability, especially when the operating field is large. Otherwise, the shares for different secret values may overwrite each other frequently. In our scheme, even when such rare events occur, they can be detected by the verifying function and the tampered values can be recovered by the recovering function. 3) Generating share addresses is on the critical path of secret sharing operations. Thus, the chosen pseudo random number generator should be efficiently implementable with a few instructions.

In our implementation, we choose a classical pseudo random generator called *Linear Feedback Shift Register*

(LFSR). We show that it can be easily implemented and has a couple of important properties. Figure 6 shows the diagram of LFSR [15].

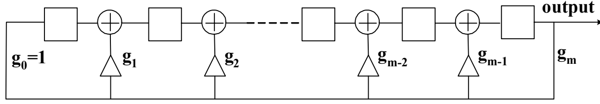

Figure 6. Diagram of LFSR.

The above implementation (Galois implementation) consists of a shift register, the contents of which are modified at every step by a binary-weighted value of the output stage. Each feedback weight g_i is either 0, meaning no connection, or 1, meaning it is fed back. Two exceptions are g_0 and g_m, which are always 1 and thus always connected [15]. LFSRs have long been used as a pseudo random number generator due to its ease of construction, long period and uniformly distributed outputs. More importantly, an LFSR has the following interesting property: an LFSR of any given size m (the number of bits in the shift register) is capable of producing every possible state during the period of $N=2^m-1$, but will do so only if proper feedback taps have been chosen. Such a sequence is called a *maximal sequence*. In other words, with a proper configuration, an LFSR of size m is able to generate all possible m bit numbers except for 0, during a period of $2^m - 1$. In our implementation, such a configuration is chosen to achieve maximal period. How to determine *maximal sequence* feedback taps is explained in [15]. Figure 7 shows an LFSR of size 3 with *maximal sequence* feedback taps. If the initial state of the register is 111, the maximal sequence generated is 101, 001, 010, 100, 011, 110, and 111. In secret sharing functions, the transformed (XORed with the process key) original address of the secret value is fed to the LFSR as seed, i.e. the initial state of the shift register. The same set of share addresses are produced in all secret sharing functions since the original address is fixed and the process key is not changed during the life time of the process.

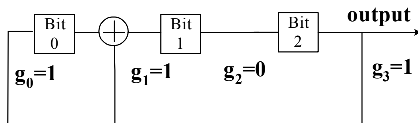

Figure 7. An LFSR of size 3 with maximal sequence taps.

The pseudo code for the LFSR random number generator is as follows. Let the *feedback weight vector G* be an m-bit binary number, so that each bit from bit m-1 to bit 0 indicates that a particular connection g_i (i from m to 1) is present on Figure 6. The bit sequence in the shift register is marked as *A*. Then the operations shown below constitute one step. The operations are very simple and can be done in a few instructions.

$$G = g_{m-1}g_{m-2}\cdots g_0$$
$$A = a_{m-1}a_{m-2}\ldots a_0$$
$$output = A >> (m-1)$$
$$A = \{[A \oplus (output\,?\,G:0)] << 1\} \,|\, output$$

The LFSR can be easily configured. The size of it is determined by the size of the memory space for the shares. Assume the share memory space is 256MB; the LFSR will be 28 bits. Next, the feedback weight vector should be set properly such that the resulting LFSR generates a *maximal sequence* regardless of the initial state. As mentioned in [15], there are many feedback weight vectors available to guarantee the generation of a maximal sequence.

3.4 Exception handling

The exceptions raised by the verifying function are most likely caused by attacks. It is also possible that the tampering is caused by the collisions of share addresses generated by the pseudo random function, although the probability is very low if the share memory space is reasonably large and the pseudo random function is good.

All the tampering to protected data, either caused by attacks or by address collisions, is handled by the exception handler. The original address of the critical data and the currently active function information are passed to the exception handler. The handler has access to a structure similar to a symbol table that records related information for all critical data. The table first records whether the critical data is identified automatically or by the user. In the former case, the default exception handler is invoked. It first dumps out the current program state including the original address of the faulty data, all machine registers, the program call stack information etc. It may also dump out critical operating system state including process control blocks, virtual memory space mappings etc. The information dumped will be used for later forensic analysis. Then it invokes the recovering function to recover the tampered data and resumes the execution of the program. In the later case, the user supplied exception handler is invoked instead, if there is one. The user supplied handler can perform application specific exception handling. With the help of the critical data symbol table, the handler knows which data is tampered and is able to react correspondingly. For example, a record in a database may be huge. Instead of protecting the whole record, the user may simply protect the master key of the record and a few critical fields. If any protected field is tampered, the user may refresh the whole record by a query to the underlying database. Another example is that the user may recalculate some dependent variables of a depended critical variable when the critical variable is tampered and recovered. In that way, strong security can be applied to the depended critical variable without excessive overhead to protect other variables derivable.

4. Security Analysis

In this section, we analyze the security of our scheme and summarize what kinds of attacks can be prevented by our scheme.

The security of our scheme does not depend on code protection. Our secret sharing implementation is assumed to be known. The security root in our scheme is the confidentiality of the random share addresses and that is achieved by the secret configuration of our random number generator (LFSR) and the secret process key. The secret sharing implementation is same for every program but the implementation takes the above two secrets as hidden parameters, which are implemented by accessing special registers. Upon program start, the processor randomly generates a secret key for the process and chooses a LFSR configuration that is able to generate a maximal sequence. The LFSR is set up according to the configuration. Moreover, as mentioned in section 3.3, the original address of the critical data is first XORed with the process key then fed into the LFSR as the seed (the initial state of the LFSR). The attacker knows neither the actual configuration of the LFSR nor the actually seed of the LFSR. Thus, the confidentiality of the share addresses generated is protected. Assume the length of the LFSR is 28-bit. The probability for the attacker to make a right guess on the actual seed in one try is 2^{-28}. In addition, a 28 bit LFSR has around 2^{20} configurations generating a maximal sequence. The attacker has to guess the right LFSR configuration too. In summary, the security of our secret sharing scheme is equivalent to the encryption strength under a 48-bit encryption key.

Our scheme can prevent the following attacks involving critical data being stolen (read) or tampered (written): 1) buffer overflows; 2) one process maliciously reading/tampering data of another using holes in access controls or its privileges; 3) certain physical attacks such as directly reading/tampering external memory with special hardware equipments. Our scheme does not stop bus snooping attacks, in which the attacker monitors the address bus to obtain the share addresses directly. How to prevent this kind of attacks is covered in [19].

To prevent buffer overflow attacks using our secret sharing scheme, the user can leave the input buffer unprotected but protect the return address following the input buffer by secret sharing. In other words, imagine the generated loads/stores to the input buffer are merely normal loads/stores, but the generated loads/stores to the return address are actually secret sharing loads/stores (assembling/scattering). By providing an oversize input, the attacker can still overflow the buffer. However, the input buffer is written using normal stores. Thus, by overflowing, the attacker tries to tamper the return address using normal stores, which is futile in our scheme since the actual return address has been secret shared and stored elsewhere. The essential idea is to treat the input buffer that could be overflowed and the critical data such as return addresses differently, which is similar to [8]. Our scheme can also protect reading/tampering attacks from other processes since each process has a different process key that drives the secret sharing algorithm. Finally, our scheme can protect simple physical attacks such as reading/writing memory directly, since critical data is secret shared and no plain text of the critical data is stored in memory.

5. Compiler optimizations

In this section, we propose several compiler optimizations to reduce secret sharing overhead in terms of performance and code size growth. An optimization flowchart is shown in Figure 8. The first optimization is secret-sharing-aware register allocation, which attempts to allocate registers such that sensitive values get higher priority to be put into registers to eliminate their secret sharing overhead. Note that under our machine model, registers are considered to be secure. We then apply rematerialization to recompute spilled sensitive values based on trusted register values instead of going through secret sharing scheme. Rematerialization is another way to avoid secret sharing overhead. Finally, secret sharing functions have to be inserted for sensitive values that have to reside in the external memory. We have several optimizations to reduce secret sharing function overhead. Since secret sharing function insertion may increase register pressure, we go back to register allocation again until no extra spills are generated during secret sharing function insertion.

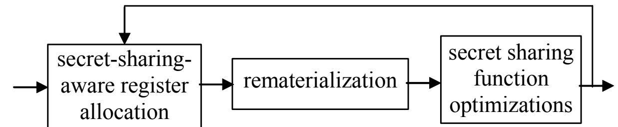

Figure 8. Compiler optimization flowchart.

5.1 Secret-sharing-aware register allocation

In our machine model, we assume registers are secure storage in the processor. Especially they are protected during context switches. Thus, if a sensitive data value spends its entire lifetime in a register, there is no need to perform any secret sharing operation on it. We modify the traditional register allocation algorithm slightly to incorporate security requirement of variables. Most register allocation algorithms, either graph coloring based or live range splitting based, have a function to calculate the spill cost of variables. The spill cost function has to be modified for sensitive variables. If in the original spill cost function, the cost of a load instruction is l, now we modify it to $nl+c$ for a sensitive variable

protected by secret sharing, in which n is the number of total shares and c is an estimation of computational overhead of secret sharing operations. Same rule applies for store instructions. In this way, sensitive variables protected by secret sharing are given higher priorities during register allocation.

5.2 Rematerialization

Rematerialization is a standard technique to trade off between spill loads cost and extra recomputation cost [17]. It can reduce secret sharing overhead in a similar way. In our case, whenever we can recompute a sensitive value based on trusted register values with a small computation cost, we should recompute it instead of going through expensive secret sharing operations. The recomputed value can still be trusted. Our algorithm considers live range extension to increase the number of variables that can be rematerialized. During live range extension, we attempt to avoid extra spills but only introduce extra move instructions.

After register allocation, all spill loads can be identified. Assume our optimization target is a spill load instruction for a value v. The rematerialization phase consists of the following steps.

1) Assume stores are definitions to a memory variable, we find out all stores, whose definitions reach the program point of v's spill load. For each such store, we find out the instructions and live ranges which compute the v before it is stored.
2) If all these live ranges reach the program point of v's spill load, v is surely rematerializable.
3) If (2) fails, we check to see if some of the live ranges can be extended to the program point of v's spill load. This may involve extra cost when we have to rename other live ranges with move instructions. For each live range to be extended, measure the register pressure between the dead point of the live range (d) and the v's spill load location (l). If at any point along any path between d and l the number of co-live values is smaller than the number of registers, mark the register pressure between d and l up by 1 and continue; otherwise, give up rematerializing the value v. In this way, we guarantee that rematerialization will not introduce extra spills, but only extra move instructions.
4) If all necessary live ranges can be successfully extended and the cost of extra move instructions and recomputation is less than the estimated secret sharing cost, this spill load can be removed and so does the secret sharing cost.

5.3 Reducing overhead of secret sharing functions

Secret sharing functions lead to significant runtime performance overhead. Our goal is to optimize secret sharing implementation to reduce program slowdown as much as possible at the same time control code growth.

<u>*Multiple instances for each secret sharing function*</u>

In the prototypes of the secret sharing functions, k and n are two parameters. However, these two parameters only have a very limited number of possible values in our implementation. To eliminate parameter passing overhead of k and n, we create different versions of the same function after static binding of k and n. Each pair of k and n gives out a different instance of the function.

<u>*Loop unrolling*</u>

There is little control flow inside our secret sharing functions. Most computation in our secret sharing scheme is done in *for* loops with very small loop bodies. As mentioned earlier, we create an instance of each secret sharing function for each k and n pairs. Once k and n are fixed, the number of iterations of each loop is decided as well. We can simply unroll *for* loops to eliminate all the loop branches to achieve better instruction level parallelism. Unrolling is preferable since k and n cannot be very large thus unrolling has better performance and little overhead. In addition, loop unrolling helps function splitting as introduced below.

<u>*Function Splitting*</u>

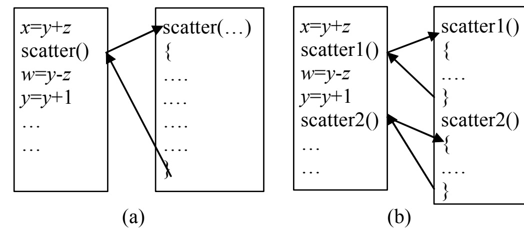

Figure 9. Function splitting.

One important observation is that secret sharing functions are independent from the original program code; therefore we should try to parallelize them. Ideally they should be scheduled together optimally so that a superscalar processor can efficiently exploit the higher instruction level parallelism embedded in the code. However, this requires secret sharing code to be inlined first, which leads to code explosion. Hereby we propose function splitting, which cuts functions into smaller pieces such that the function is executed piece by piece interleaved with the execution of the original program instructions. For example, Figure 9.a shows a piece of code calls the scattering function. As mentioned earlier, each secret sharing function contains almost sequential code after loop unrolling. We can split the function body in the middle and the function can be executed piece by piece--Figure 9.b. After splitting, the original code and

the secret sharing code are interleaved during execution, achieving certain level of parallelism.

However, in this way the function call overhead is increased, since one function call becomes two or more. Fortunately, the function call overhead can be made very small. As we mentioned, we create multiple instances of a secret sharing function to reduce parameter passing overhead. Also, we tweak register allocation algorithm to reduce caller/callee saved register overhead as will be explained shortly. Moreover, we only split a function into several pieces. The number of instructions in each piece should not be too large, otherwise the interleaving effect diminishes. Also, it should not be too small, otherwise it would incur too much function call overhead. In our current implementation, we make each piece contain approximately 16 instructions. For each secret sharing function, its pieces should be scheduled in sequence and all pieces have to be scheduled before the output of the function is used.

To reduce caller/callee saved register overhead during a call to a secret sharing function piece, we designate several registers for secret sharing functions. These registers have lower priority to be allocated in the original code. Thus, in many cases caller/callee saved register overhead is eliminated completely. When inserting calls to the secret sharing functions, we attempt to find program points that incur less caller/callee saved register overhead. The procedure for insertion of split functions is as follows.

1) For each insertion, due to dependencies, there are only a limited number of program points where secret sharing function pieces can be inserted. All such program points are identified.
2) The caller-saved register overhead is calculated for each program point.
3) All possible insertion combinations are attempted and the one with minimal overall cost is chosen.

There are a few issues we want to address further. First, the number of pieces of a secret sharing function after splitting is typically small, therefore an exhaustive search is affordable. Second, due to control flow split and merge, we might need to insert duplicated function pieces along multiple paths. Our algorithm makes sure that along each path the same sequence of function pieces are encountered.

6. Experiments and results

In this section, we evaluate the cost of our secret sharing scheme and the effects of our optimizations. Our target processor is an Alpha 21264 processor. There is no standard security benchmark suite publicly available. Thus we choose all SPEC2000 integer programs as benchmarks. For each benchmark, we apply our secret sharing scheme and regenerate the Alpha binary. The newly generated Alpha binaries are fed into the SimpleScalar toolset [5] for performance measurement. SimpleScalar is a widely used cycle-accurate processor simulator that can give in-depth performance results. A simulator is used so that we can experiment with different architecture setups conveniently. Each benchmark is executed to finish with the reference input data set. The default parameter for our secret sharing scheme is a typical setup of k=3, n=5. The field on which secret sharing is operated is determined by a 65 bit prime number. So any data type less than or equal to 64 bits can be protected without further processing. The share address space is a 256MB buffer allocated in the heap. Finally, we have not fully implemented the exception handling mechanism to recover tampered values yet since it involves modifications to the OS. We will not measure the complete recovering cost in the experiments.

We perform two sets of experiments with different data protection coverage. In the first experiment setup, we only protect critical data that can be automatically identified without application specific knowledge, such as stack return addresses, jump tables and virtual function tables. In the second experiment setup, besides the data mentioned above, we also protect critical data specific to the application that is specified by the user. By default, 1/5 of the application data is protected by secret sharing and all optimizations are enabled. Before we present results of our experiments, we first show in Table 1 the computational cost (in cycles) of each secret sharing function in isolation under three typical k and n setups. The cycle numbers are measured using SimpleScalar simulator [5] and they are pure computation cycles and no memory latency is counted in.

Table 1. Cycle cost of secret sharing functions.

	scatter	assemble	verify	recover
k=2, n=3	23	15	18	51
k=3, n=5	36	17	20	180
k=4, n=7	53	19	22	693

6.1 Protecting automatically identified critical data

In this section, we focus on the first experiment setup in which only critical data automatically identified by the compiler is protected by our secret sharing scheme.

Figure 10 shows the binary size growth after automatically identified critical data (stack return addresses, jump tables and virtual function tables etc.) is protected by our secret sharing scheme. The sizes in bytes of the original binary (labeled original) and the binary with optimized secret sharing (labeled SS) are shown in the y axis. We observe that the code size growth is minor, from 0.96% to 2.4% with an average of only 1.7%. The reason is that secret sharing functions are invoked through function calls and no function inlining is performed. In that way, there needs to be at most a couple

of copies of secret sharing functions due to versioning of secret sharing functions.

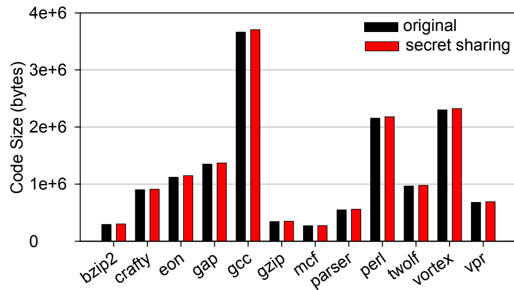

Figure 10. Binary size growth (automatic protection).

Figure 11 shows the performance (inverse of execution time) for our secret sharing scheme normalized to the baseline without secret sharing protection. Three configurations are shown: k=2, n=3; k=3, n=5; k=4, n=7. The average performance degradation under these three configurations is 3.2%, 3.9%, and 4.7% respectively. The observation is that the performance degradation is generally small, even strong protection is applied. One important reason is that our implementation of secret sharing is highly optimized and pre-computation reduces computational overhead greatly. Another reason is that the amount of compiler identified critical data is quite limited and the performance degradation largely depends on the amount of data protected. Since the overhead is very small when only compiler identified critical data is protected, we finish this experiment setup quickly and focus on the tougher case.

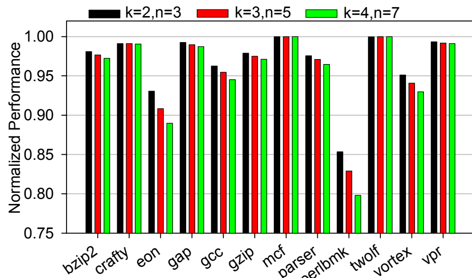

Figure 11. Performance degradation (automatic protection).

6.2 Protecting application-specific critical data

In this section, we focus on the second experiment setup, in which user-identified application-specific critical data is also protected. Since there is no publicly available standard security benchmark suite, we still use SPEC2000 integer programs as our benchmarks. It is hard to evaluate the impact of secret sharing scheme to the binary size and performance under this experiment setup. The reason is that the application specific critical data is specified by the user according to the semantics of the program and there is no standard way to do that. However, normally only a small percentage of data is security sensitive. A good example is a personal data record. In the record, there is sensitive information such as SSN, date of birth, banking account numbers etc. There is also non-sensitive information such as the home address and telephone numbers etc., which could be easily obtained by a lookup in yellow pages. In our experiments, we try different percentages of protection and examine the resulting binary size and performance overhead. For each data protection percentage, we randomly pick up that percentage of application data to apply secret sharing scheme on it. Although which set of data is protected has less impact to binary size, it may have significant impact to runtime performance since different data has different access pattern/frequency. To make our results more reliable, we pick up ten sets of data randomly for each protection percentage, then take the average as the final result.

Figure 12 shows the impact to binary size with different percentages of application data protected by optimized secret sharing scheme. Normalized binary sizes are shown. We experiment five different configurations, with 1/10, 1/5, 1/3, 1/2 and all application data protected respectively. The average binary size growth is 1.6%, 8.4%, 15.3%, 24.4%, 35.8% and 70% respectively. It is apparent that the binary growth depends on the protection percentage. The growth is significant when a large percentage of application data is protected due to the instructions to invoke secret sharing functions. However, as discussed, normally only a small percentage of application data is critical/sensitive.

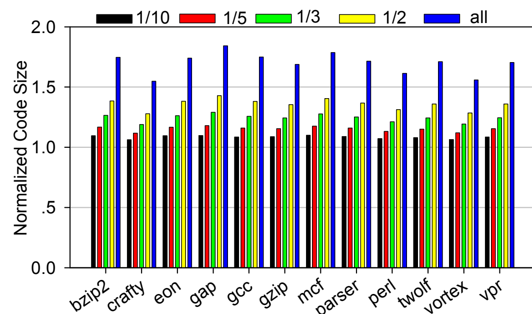

Figure 12. Binary size growth (user protection).

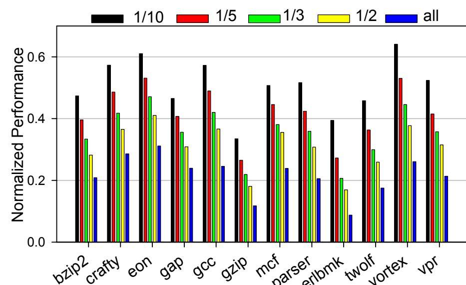

Figure 13. Performance degradation (user protection).

Figure 13 shows the impact to performance with different percentages of application data protected by optimized secret sharing scheme. Performance numbers are normalized to the baseline configuration without secret sharing protection. We experiment five different configurations. The average performance degradation when 1/10, 1/5, 1/3, 1/2 and all application data protected is 49.4%, 58.1%, 64.4%, 69.1% and 78.4% respectively.

The performance degradation is significant, especially when a large percentage of data is protected. However, please note that our scheme is a software based scheme and security normally costs a lot. In [11], the authors show that even with hardware support, using encryption to protect data confidentiality leads up to 50% performance degradation. In [12], the authors show that even a hardware implementation of the Merkle tree integrity checking scheme could degrade performance by another 30% on average. Moreover, our scheme provides not only confidentiality and integrity, but availability, which is never done by the proposed hardware schemes. In summary, we feel that the performance degradation of our scheme is acceptable, especially when the developer is able to identify a small set of data to be protected. If with additional hardware support, we believe the performance degradation will become minor.

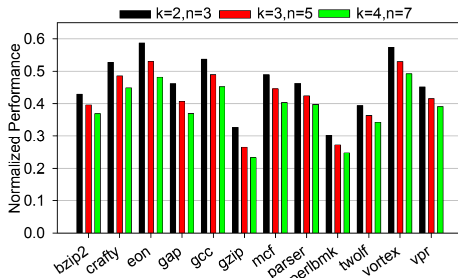

Figure 14. Protection strength sensitivity study (user protection).

Figure 14 shows the impact of different k and n values to performance. Default configuration is used except that k and n values are varied. All optimizations are enabled. As discussed earlier, higher k and n will lead to greater performance overhead. Three different configurations are experimented, i.e., k=2, n=3; k=3, n=5; k=4, n=7. K=3 and n=5 is our default configuration. Our results show that when k=2 and n=3, the average performance degradation is 53.8%; when k=3 and n=5, the average degradation is 58.1%; when k=4 and n=7, the average degradation is 61.4%.

Figure 15 shows the effects of our optimizations. Performance numbers are normalized to the case in which no optimization is enabled. We show three cases: 1) only secret-sharing-aware register allocation is enabled; 2) secret-sharing-aware register allocation and rematerialization are enabled; 3) all optimizations (secret-sharing-aware register allocation, rematerialization and optimizations to secret sharing functions implementation) are enabled. With secret-sharing-aware register allocation, performance is improved 5.1% on average over the case without optimization. With the help of rematerialization, the performance is improved by 8.9% on average. When all optimizations are enabled, the performance is improved by 10.5% on average. Thus, our optimizations improve performance significantly.

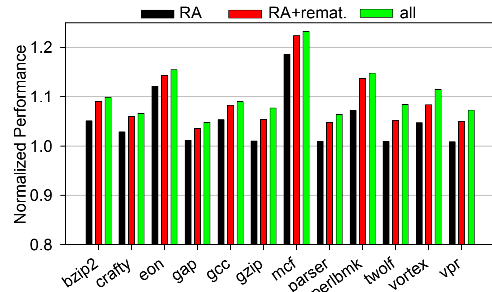

Figure 15. Effects of optimizations (user protection).

Finally, Figure 16 shows compilation time cost. We use MachSUIF compiler infrastructure [20] and all the optimizations are implemented in the register allocation pass of MachSUIF compiler. Time spent in the register allocation pass normalized to the case without optimizations is shown. On average, our optimizations increase the time on the register allocation pass by 26%.

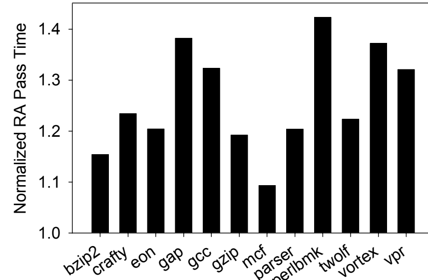

Figure 16. Time cost of register allocation pass (user protection).

7. Related work

Buffer overflow is the source of most attacks and many solutions have been proposed to protect against it. [6] is a compiler technique which prevents changes to active return addresses by either detecting the change of the return address before the function returns, or by completely preventing the write to the return address. [7] proposes another compiler solution to protect code pointers from buffer overflows by encrypting pointers when stored in memory and decrypting them when loaded to registers. [8] studies hardware support for code pointer encryption to achieve better security and performance.

In this work we focus on a much more general

problem – how to provide confidentiality, integrity and availability to all kinds of security sensitive data including return addresses and code pointers. We propose a secret sharing [3] based compiler solution to achieve the above three properties all together, rather than tackle them in isolation. Secret sharing has been widely used in application domains such as key management [9] and secure distributed file system [10]. However, our work is the first attempt to apply secret sharing to protect application data.

Recently, hardware based solutions to protect data confidentiality and integrity have been proposed [11][12][13][14], which need special crypto hardware support inside the processor and achieve better security and performance. However, first, neither of them considers tamper recovery; second, the architecture of current popular processors such as Pentium is radically different from the one proposed in [11]. It is not realistic for the proposed secure architecture [11] to be adopted in the near future. So it is critical to design a software based solution with minimum hardware support which requires little hardware modification.

8. Conclusion

It is extremely important to secure critical computer systems. In this work, we focus on protecting sensitive data of software. Security implies several important properties including confidentiality, integrity and availability (intrusion tolerance). We develop a secret sharing based compiler solution to achieve these properties altogether, rather than tackling them one by one as in previous approaches.

The implementation of our secret sharing scheme is carefully crafted to achieve low overhead. We further propose several compiler optimizations such as secret-sharing-aware register allocation, rematerialization etc. to reduce the cost of secret sharing further, making our scheme a practical solution in a high performance machine. Through experiments, we show that the performance overhead can be made reasonably small (58%) to protect a significant amount of sensitive data (20% of application data).

Acknowledgements. We are grateful to Dr. Michael Smith and anonymous reviewers for their feedback. Their comments were very helpful in revising the contents and presentation of the material.

References

[1] Aleph One, "Smashing the Stack for Fun and Profit", Phrack volume 7, issue 49.

[2] Federal Trade Commission, "Federal Trade Commission – Identity Theft Survey Report", September 2003.

[3] A. Shamir. How to Share a Secret. Communications of the ACM, 22(11): 612- 613, November 1979.

[4] R.J. Anderson, "Security Engineering", John Wiley and Sons publishers, February 2001.

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

[6] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks", In Proceedings of the 7th USENIX Security Symposium, pages 63--78, January 1998.

[7] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In 12th USENIX Security Symposium, Washington DC, August 2003.

[8] Nathan Tuck, Brad Calder and George Varghese, "Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow", 37th International Symposium on Microarchitecture, Dec. 2004.

[9] D. Boneh, M. Franklin, "Efficient generation of shared RSA keys", in Proceedings Crypto' 97, pp. 425--439.

[10] Dabek, F., Kasshoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with CFS. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01), Banff, Canada, Oct. 2001.

[11] 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.

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

[13] J.Yang, Y.Zhang, L.Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering," *International Symposium on Microarchitecture*, Dec. 2003.

[14] 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, Dec. 2003.

[15] LFSR reference. http://www.newwaveinstruments.com /resources/articles/m_sequence_linear_feedback_shift_register_lfsr.htm

[16] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, P. Markstein, "Register allocation via coloring", *Computer Language*, vol.6, pp.47-57, 1981.

[17] Preston Briggs, Keith Cooper and Linda Torczon, "Improvements to Graph Coloring Register Allocation," *ACM Transactions on Programming Languages and Systems*, 1994.

[18] Y-M. Wang, Y. Huang, K-P. Vo, P-Y. Chung, and C. Kintala. Checkpointing and its applications. In 25th International Symposium on Fault-Tolerant Computing, pages 22--31, Pasadena, CA, June 1995.

[19] X. Zhuang, T. Zhang, S. Pande, "HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus", ASPLOS XI, Boston, MA., Oct 2004.

[20] Mach-SUIF Backend Compiler, The Machine-SUIF 2.1 compiler documentation set. Harvard University, Sep. 2000.