InfoShield: A Security Architecture for Protecting Information Usage in Memory

Weidong Shi Joshua B. Fryman[1] Guofei Gu
Hsien-Hsin S. Lee Youtao Zhang[2] Jun Yang[3]

School of Electrical and Computer Engineering, College of Computing, Georgia Tech
[1]Programming System Lab, Corporate Tech. Group, Intel Corporation
[2]Dept. of Computer Science, University of Pittsburgh
[3]Dept. of Computer Science and Engineering, University of California, Riverside

leehs@gatech.edu {shiw, guofei}@cc.gatech.edu joshua.b.fryman@intel.com zhangyt@cs.pitt.edu junyang@cs.ucr.edu

ABSTRACT

Cyber theft is a serious threat to Internet security. It is one of the major security concerns by both network service providers and Internet users. Though sensitive information can be encrypted when stored in non-volatile memory such as hard disks, for many e-commerce and network applications, sensitive information is often stored as plaintext in main memory. Documented and reported exploits facilitate an adversary stealing sensitive information from an application's memory. These exploits include illegitimate memory scan, information theft oriented buffer overflow, invalid pointer manipulation, integer overflow, password stealing trojans and so forth. Today's computing system and its hardware cannot address these exploits effectively in a coherent way. This paper presents a unified and lightweight solution, called **InfoShield***, that can strengthen application protection against theft of sensitive information such as passwords, encryption keys, and other private data with a minimal performance impact. Unlike prior whole memory encryption and information flow based efforts, InfoShield protects the usage of information. InfoShield ensures that sensitive data are used only as defined by application semantics, preventing misuse of information. Comparing with prior art, InfoShield handles a broader range of information theft scenarios in a unified framework with less overhead. Evaluation using popular network client-server applications shows that InfoShield is sound for practical use and incurs little performance loss because InfoShield only protects absolute, critical sensitive information. Based on the profiling results, only 0.3% of memory accesses and 0.2% of executed codes are affected by InfoShield.*

1. INTRODUCTION

The leakage of sensitive information in network servers can result in serious consequences. First of all, any disclosure of credential information such as a login account with password can assist adversaries in gaining privileged accesses to computing facilities. Also, theft of user information such as credit card numbers, social security numbers, and other personal information is a major concern of companies offering online services. The loss in revenue due to cyber vandalism and information theft is in billions of dollars annually [6]. There are many techniques that can be applied by a remote or local adversary to compromise data privacy. These techniques include, but not limited to, password stealing worms/trojan horses [32, 27], memory scans using either legitimate or malicious software [15], theft or destruction of private or system data via buffer overflow exploits [20], as well as stealing sensitive information through altering memory pointers, data offsets, or array indices [20].

The causes of data privacy violation are diverse, so are the solutions. In fact, there are very few studies in the past that directly address the problems from the aforementioned attacks. In many cases, data privacy often comes as a by-product of safe programming practices using either security analysis tools or an intrinsically safe source language. For example, special compiler and program analysis tools [5, 16, 4] were developed to reduce the risks of buffer overflow and memory reference errors. This approach indirectly mitigates the risk of disclosing sensitive data through these types of exploits. Two particular research areas that directly address the issue of data safety are *access control* and *information flow analysis* [7, 29, 10, 1]. As we will show later, traditional access control approaches do not provide sufficient protection on information safety because the OS-based access control is too coarse-grained. On the other hand, using static information flow analysis to ensure privacy and security of information is sometimes too restrictive [21] for real applications where sharing or disclosing information is mandatory and frequent.

In this paper, we present a new approach, termed *InfoShield*, to protecting the privacy of sensitive information based on *information usage*. As can be readily seen from Internet viruses and spyware, misuse of information is one major threat to information security. Many attacks stem from abnormal or unauthorized usage of information. For example, a password or encryption key must be strictly used for access authentication by the designated functions based on a specific control flow. Any other usage of the information, e.g. memory scan or following a different control flow path, should be considered unsafe and prohibited to prevent a security breach. Specifically, the major contributions of this paper are:

- Addressing the issue of information usage safety and its implications to defending real attacks to information privacy.
- Presenting a novel architecture called InfoShield that enforces proper usage of sensitive data according to program definition or semantics at runtime.
- Proposing lightweight hardware solutions to facilitate and accelerate the process of information usage protection.

The unique characteristics of information usage based protection on data privacy are:

- **Improved data privacy.** The notion of information usage safety is proposed based on vulnerability analysis and exploits to address real threats. Particularly, the usage verification and authorization are performed at runtime on-demand due to the dynamic nature of information usage.

- **Increased enforceability.** The proposed InfoShield is easily enforceable for real software systems where information is frequently shared or disclosed. It leaves the decision of what information can be shared or disclosed to the software developers, therefore avoiding the overprotection problem associated with traditional information flow based protection. It enforces protection on usage of sensitive information by ensuring that the data can only be used in a specific way by the designated program according to pre-defined program semantics.
- **Optimized performance.** InfoShield guarantees proper usage of information at a fine granularity (down to each word in memory). It verifies the usage of information at runtime by replacing some LD/ST operations with security-aware equivalents. Using architectural support, InfoShield can achieve real-time protection of information usage with negligible performance loss.
- **Composable security.** The proposed protection on information usage is orthogonal to other language based schemes on program security. It provides composable security by allowing it to be combined with other techniques such as information flow based protection. For example, static information flow based analysis can be applied first to ensure that a program is written properly and there is no leakage of private information based on program semantics. Then, information usage protection guarantees that information is used exactly as defined, precluding any undesired disclosure of sensitive data.

The remainder of the paper is organized as follows. In Section 2, we discuss threats on information privacy caused by information misuse. Section 3 presents the InfoShield architecture. Section 4 and Section 5 examine security issues and evaluate the performance of InfoShield. Section 6 is an overview of related work, and finally Section 7 concludes the paper.

2. THREAT MODELS

There are many software exploits that can result in disclosure of sensitive information. These exploits can be either launched by remote attackers or local access adversaries.

2.1 Attacks on Information

2.1.1 Memory scan

Memory scanning is one of the most straightforward ways to search for sensitive information stored in either application or kernel memory. If adversaries know the whereabouts of the secret information, their job would be much easier. If adversaries have local access, they can use memory dumping tools that often can access higher privilege memory [15]. For remote exploits, techniques such as memory resident worms, Trojan horses, or malicious dummy kernel drivers can be used. After loading in memory, such malware can periodically scan application or even kernel memory for sensitive information. When found, the malware can send the information out without the knowledge of the data owner. As shown in [15], many important data can be easily recovered by a simple keyword-based scan of memory. For example, the iDefense security advisory [11] documents a number of memory scan attacks for logon credentials applicable to popular network clients including PuTTY ssh2 client, SecureCRT, and AbsoluteTelnet.

There are three basic memory scan techniques that allow adversaries to identify secrets in an application's virtual memory. First, they can search for a pivot string such as "password" or "ssh-connection." The real password and encryption key is often stored at a location with fixed offset from the pivot string [11]. Second, adversaries can run a local copy of the same system and reverse-engineer the likely locations where sensitive data may be stored [15]. Third, entropy-based analysis can be used to discover encryption keys stored in random places of a server memory [28]. The effectiveness of this technique to break encryption keys has been demonstrated by a commercial key finding tool developed by nCipher [35].

2.1.2 Invalid input

Another technique is to manipulate input data to make the victim application misbehave, disclosing sensitive data (in)voluntarily. For example, the Linux kernel allows a local adversary to obtain sensitive kernel information by gaining access to kernel memory via vulnerabilities in the /proc interfaces. One documented vulnerability with 32- to 64-bit conversions in the kernel [20] allows insecure modification of file offset pointers in the kernel using the file API (such as open and lseek). Through carefully orchestrated manipulation of file pointers, a /proc file can be advanced to a negative value that allows kernel memory to be copied to user space. These vulnerabilities allow a local unprivileged attacker to access segments of kernel memory which may contain sensitive information. A similar example is a flaw in the FreeBSD process file system (GENERIC-MAP-NOMATCH) [20]. In the uiomove system call, a local attacker can set the uio_offset parameter of struct uio to extremely large or negative values, to cause the function to return a portion of the kernel memory, e.g. terminal buffers, which could contain a user-entered password. Aside from the OS kernel itself, device drivers and other vendor-centric software also reside in the kernel space. Vulnerability of these modules can also be exploited. For instance, the Linux e1000 Ethernet card driver had a flaw that unbounded input can be redirected as an input for the *copy_to_user* function, causing kernel memory to be returned (CAN-2004-0535) [20].

2.1.3 Buffer overflow

Buffer overflow [2], a traditional exploit technique, allows attackers to overwrite data in an application's stack or heap with arbitrary data or malicious code. The injected code, if executed, helps the attacker gain access to sensitive information. Adversaries can also elevate their privilege level by running injected code through buffer attacks. A specific form of buffer overflow is the format string attack [26] that exploits any vulnerability of a format specifier (e.g., "%n") in standard C functions. The existence of the format string attack in some cases allows sensitive information to be disclosed without code injection. One example is the documented Mac OS X *pppd* format string vulnerability (CAN-2004-0165) [20]. When *pppd* receives an invalid command line argument, it will eventually pass it as a format specifier to vslprintf(), which can be exploited to read arbitrary data out of *pppd*'s process. When the system is used as a PPP server under certain circumstances, it is also possible to steal PAP/CHAP authentication credentials by exploiting the format string vulnerability without code injection.

2.1.4 Worms and Trojan horses

A large number of worms and Trojan horses are concocted to compromise users or system information [32, 27]. As studies indicate, approximately 90% of Trojan horses found in circulation today are from online services. A significant number of them try to steal sensitive information such as login IDs or passwords. Examples of such Trojans include W32/Eyeveg-B, VBS/LoveLetter.bd, BadTrans.B, and Lirva [27]. These respectively attack online-banking customer accounts, send compromised information to an email address, or steal password by crawling through ICQ, email, or peer-to-peer file sharing systems. Lirva can even disable antivirus and security applications.

The aforementioned exploits on information privacy achieve their goal through misuse of information or through induced software misbehavior. Sensitive information such as passwords and crypto keys should be strictly used only by the designated codes rather than a third party software such as memory scan tools. Memory reference

manipulation through invalid input or buffer overflow are obviously abnormal information usage. Note that these kind of exploits cannot be prevented by a simple shutdown of all the output channels. In many cases, the victim software is supposed to disclose certain information. However, through maliciously induced changes on pointers or memory offset, other (sensitive) information is disclosed instead.

2.2 Assumption of Attack Models

Note that InfoShield is designed for an enterprise computing environment, where theft of sensitive user or system information through any attack is a major concern. However, our simple implementation is useful for any domain that has security concerns, such as consumer products. InfoShield is not for countering sophisticated physical or side-channel attacks that require a skill set that exploits bus traffic using a logic analyzer or oscilloscope. We assume that the computing platforms are physically secured. Furthermore, we are only concerned about certain critical information such as credit numbers, social security numbers, passwords, login accounts, encryption keys, etc. Protecting the usage of all data is often unnecessary and too costly. Also note that InfoShield assumes integrity protection on application code, namely, semantics of the application code cannot be arbitrarily changed. This assumption is in fact required by almost all the runtime based information security schemes including dynamic tracking of information flow [31] and proof carrying program execution [22, 23]. Simple solutions based on trusted computing systems such as LaGrande Technology can have programs signed or certificated. Integrity of the signed program code can be verified before it is executed and memory pages of the verified codes are marked write-protected. More rigorous protection on program integrity such as [9] can be also employed to prevent even runtime tampering with code integrity. Code signing itself does not provide protection for data privacy.

3. INFOSHIELD ARCHITECTURE

Under the concept of information usage safety, what can be disclosed or what can be shared is determined by the application and programmers. It is assumed that the program was properly written and audited, such that it shares or discloses only information that should be shared or disclosed. Then InfoShield enforces at runtime that sensitive information is used only in the way defined during program development.

3.1 Information usage safety

In this subsection, we compare the differences among several methods for information safety and address the necessity of information *usage* safety. As mentioned in Section 1, access control provides only a limited coarse-level data protection in memory. Attacks such as injected code, memory scan, or input manipulation cannot be prevented by simple user level or OS based access control. Another concept is *information flow safety* [1, 7, 10, 29]. According to this notion, information is labelled based on its privilege or security level, and an information flow tracking mechanism assures that privilege information does not flow to a channel with lower privilege or lower security levels. Information flow safety is a powerful concept but sometimes becomes too restrictive. For example, an encryption key is considered as a high security level item. Assuming that a user wants to send low privilege information, but prefers to have it encrypted first, then the encrypted data will carry information about the encryption key. The consequence is that the resulting encrypted information should also be considered as a high security and thus unsafe to be shared or disclosed. An alternative concept, called *computational safety* is used to address this problem. According to this concept, a piece of information (although carrying high privilege information) is considered safe to be disclosed if it is computationally infeasible to extract the sensitive information from the disclosed data. Based on this concept, disclosing or sharing encrypted data is a safe operation if the encryption algorithm can ensure computational safety. Some efforts have been made in the past on finding a middle ground between these two concepts to avoid the problem of overprotection based on traditional information flow analysis [33, 17]. Table 1 summarizes the differences of the three information safety concepts.

However, information flow safety and information usage safety are complementary to each other since they address different aspects of information security. The former can be applied to decide what information is safe to be disclosed by program semantics, while the latter guarantees that no information misuse occurs according to the defined semantics. Note that there are many subtle differences between information flow and information usage. For example, a program may have two components, A and B, with each one disclosing a piece of information, a and b, at the same security level. According to information flow safety, A can also disclose information b. However, based on information usage safety, A cannot disclose b if this operation is not defined according to the program semantics. This prevents malicious exploits on pointers of A to make them point to b.

3.2 Protection on information usage

Here we present a protection scheme for information usage security that uses real-time authorization codes in the application. The necessary program alterations can be generated directly by a security enhanced compiler or by a separate information usage security analysis tool which does binary rewriting.

The basic idea of real-time protection of information usage are: 1) permission to use sensitive information is granted dynamically, exactly according to application semantics; 2) usage of information is dynamically verified to ensure that it conforms with the defined usage; 3) improper access attempts cannot compromise data confidentiality. Integrity of the application information usage is verified throughout the program execution. At the micro level, InfoShield guarantees that:

- Sensitive information is used only in the order defined by the application.
- Sensitive information is used only by the instructions that must use them.
- The user is assured that no misuse of information occurs if no exceptions are raised.

Our proposed implementation of InfoShield defines new architectural instructions for supporting information usage safety. These new operations combine regular access instructions with additional functionality for tracking security state. InfoShield will only protect usage of information in memory, which is sufficient for countering most of the exploits described in Section 2.

Conceptually, the implementation of information usage protection is simple. Consider a hypothetical example where an encryption key is created and then used. At the declaration of the encryption key or its pointer, the programmer annotates the source code that the contents of the key are sensitive. When the storage of the key is created, (e.g. by `malloc`), the compiler records the returned address with the annotation indicating that it is an address for sensitive data. Prior to passing the pointer to the surrounding code for proper handling, the compiler injects additional instructions to *guard* the data contents. In a separate hardware table, this address of the key is entered along with the PC of the *next* instruction that can access the contents of this address. Every load/store operation checks this table to ensure that no access occurs when the PC is not the designated next-access PC by the compiler. Such violations raise an exception bit for later handling.

Each attempt to use the sensitive information must come from an authorized instruction. Since each instruction authorizes the next in-

Information flow safety [1, 7]	Computational safety [33, 17]	Information usage safety (InfoShield)
The encrypted result is generated using a secret key. All the bits of encrypted result carry details of the key and are considered un-safe to be disclosed	The encrypted result is computationally safe to be disclosed. It is not feasible to extract the key from the encrypted data.	The encrypted result is safe to be disclosed if it is based on correct execution of the function and there is no misuse of the key.

Table 1: Comparison of Information Safety Models

struction dynamically and instruction memory is read-only, no violation can occur. Since every load/store operation must check the security address table, and only compile-time authorized instructions can read/write the data, all non-Trojan attacks are effectively blocked. The only vulnerability is the initial registration of an address to be guarded. However, any preemptive capture of the protected data location will either result in the proper guard instruction failing, or the proper guard instruction taking control over the next-authorized PC field. In either case, no violation of the security occurs such that sensitive information is revealed. The intricacies of handling function calls, register spills, garbage collection, and so forth are detailed later.

3.2.1 Instruction extensions

The security-aware instruction extensions we propose require a limited hardware support in the processor along the lines of a primitive cache or register file with CAM lookup capability as shown in Figure 1. Primarily, a structure that can be indexed like a register file for secure address testing is required. However, the address field in the structure must also be searchable like a CAM. While a full implementation is possible without this unit, performance would be substantially degraded. An overview of the instruction extensions is shown in Table 2, where the fields referenced are from Figure 1.

	Valid	VAddr_Low	VAddr_High	NextPC_Low	NextPC_High
SR_0					
SR_1					
SR_{N-1}					

Figure 1: Security-aware Register (SR) Hardware Table.

There are three principle design axioms that ensure correct operation with negligible performance impact: (a) each Security-Aware (SAx) operation is atomic; (b) programs, including the OS, are in read-only memory pages; and (c) while LD/ST operations may be common, LD/ST operations to sensitive data are rare. To walk through an example, Figure 2(a) is a simple C program snippet, with Figure 2(b) a pseudo-assembly listing for (a). An extension to gcc provides the "secure" attribute on line 1.

In the assembly listing, after the instruction at 0xA004 has executed, no interception of the protected data is possible. Any attempt to take control over a later instruction requires that no prior security aware instruction executed properly. Similarly, any attempt to divulge memory through stack or pointer manipulation will trigger an exception to the application.

Under InfoShield's threat model assumption, a hacker cannot directly modify the original software code (protected with code signing, runtime integrity check and execution only pages). But hackers can send invalid input or hijack control flow. All the SAx instructions of an application form a chain of information usage authorization. Protected sensitive data will not be disclosed through control hijack in the middle of the chain because it violates semantics of SAx instructions. A hacker may insert SAx instruction ahead of an SAx instruction chain. But those illegal SAx instructions will fail the first legitimate SAP instruction, thus no sensitive information will be revealed. Also note that applications always store sensitive data to a memory location after a SAP that declared that location. If a SAP instruction cannot find a matching entry in the SR table, it means that it is the first SAP instruction of a SAx instruction chain. In this case, the first SAP's PC is not checked.

```
                                                 .org   $0100
_attribute secure long long *key     0x0100     .word  key

                                                 .org   $1000
                                     0x1000

key = malloc(sizeof(long long));     0xA000     call   _malloc          ; r1 = ptr
                                     0xA004     add    r2, r1, #8       ;securing 8 bytes
                                     0xA008     mov    r3, #0xB00C      ; start of next-valid PCs
                                     0xA00C     mov    r4, #0xB014      ; end of next-valid PCs
                                     0xA010     sag    r0               ; get next free security slot
                                     0xA014     sap    r0, r1, r2, r3, r4  ; enable protection
                                     0xA018     mov    r7, r1           ; r7 is the beginning of the
                                                                        secure data
                                     0xA01C     push   r0               ; save r0 for later

                                     0xAFFC     pop    r0               ; recall our slot
*key = srand();                      0xB000     call   _srand           ; r1 = random data
                                     0xB004     mov    r2, #0xC008      ; start of next-valid PCs
                                     0xB008     mov    r3, #0xC00C      ; end of next-valid PCs
                                     0xB00C     st     r1, [r7]         ; write to the key slot
                                     0xB010     sas    r0, r2, r3       ; slide protection window
                                     0xB014     push   r0               ; save the slot

                                     0xBFFC     pop    r0               ; recall our slot
                                     0xC000     mov    r2, #0xD020      ; prepare next valid PC block
                                     0xC004     mov    r3, #0xD0A0      ; big switch statement...
                                     0xC008     ld     r1, [r7]         ; secure read key val
                                     0xC00C     sas    r0, r2, r3       ; slide the protection window
                                     0xC010     push   r0               ; save our slot
err = encrypt ( data, *key);         0xC014     call   encrypt          ; make call; r1 S bit set!
              (a)                                                (b)
```

Figure 2: (a) Example of C program snippet (b) corresponding security aware assembly code.

3.2.2 Non-restrictive addressing

The security aware extensions (SAG, SAP, etc.) place no restrictions on addressing modes. In particular, there are no specific instructions to load or store data at protected addresses. This is achieved by checking *every* load/store operation against the protected addresses in the SR table by testing the effective (virtual) address. A mechanism for removing this check from the critical path is presented later.

The result of this model is that the security aware extension instructions only manipulate security information, and not data – with the exception of the SAM instruction. The need for the SAM instruction is presented in the garbage collection analysis below. Therefore, all of the standard memory addressing modes – direct, register offset, and so forth – work without modification. Only at retirement time will the security check results be used to accept or reject the results of any load/store operation.

3.2.3 Function calls

Protection of information usage across function calls is also supported. There are two basic methods for handling argument passing. If all programs (and programmers) adhere to a convention that only the address of a sensitive data location is passed as an argument, never the contents, then no further architectural support is necessary. However, considering the large volume of existing software passing arguments by registers for which simple recompilation will not repair, as well as the inability to ensure that programmers always follow safe practices, additional architectural changes will be needed to handle secure usage regardless of programmers' behavior.

First, we extend each register beyond its basic data and control state to include a security aware flag. The flag is an indicator that this specific register was read or written from a registered security aware region. Any other register derived from this register has its security

Instruction	Behavior	Purpose
SAG Rd Set Address Guard	$\exists\, n \,\&\, SR[n].Valid == 0 \Rightarrow$ $\quad RF[Rd] \leftarrow n$ $\quad SR[n].Valid \leftarrow 1$ else $CCR[SX] \leftarrow 1$	Get the first free SR table entry and return else throw an exception
SAP Rd, Ral, Rah, Rpl, Rph Set Address Protection	$\forall\, n\,,\ \exists\, m \in\ ((RF[Ral]..RF[Rah])$ $\quad ((SR[n].valid == 1)\ \&$ $\quad (SR[n].VAl\ \leq m \leq\ SR[n].VAhi))\ \Rightarrow$ $\quad Err \leftarrow Err \mid SR[RF[Rd]].Valid == 0 \mid$ $\qquad \neg[(PC \geq SR[n].PClo) \mid (PC \leq SR[n].PChi)]$ $CCR[SX] \leftarrow Err$ $SR[RF[Rd]].PClo \leftarrow RF[Rpl]$ $SR[RF[Rd]].PChi \leftarrow RF[Rph]$ $SR[RF[Rd]].Valid \leftarrow 1$ $SR[RF[Rd]].VAlo \leftarrow RF[Ral]$ $SR[RF[Rd]].VAhi \leftarrow RF[Rah]$	Verify that this PC is in the permitted group to setup target protection; if no Err, proceed to set up the valid code window that can access a memory block [Ral:Rah] and set the next-valid instruction PC range to [Rpl:Rph] in SR table
SAS Rd, Rpl, Rph Secure Address Shift	$Err \leftarrow (SR[RF[Rd]].Valid == 0) \mid$ $\quad (PC \leq SR[RF[Rd]].PClo) \mid (PC \geq SR[Rf[Rd]].PChi)$ $CCR[SX] \leftarrow Err$ $SR[RF[Rd]].PClo \leftarrow RF[Rpl]$ $SR[RF[Rd]].PChi \leftarrow RF[Rph]$	Verify this PC can slide the next-instr window; if ok, set up the valid code window that can access a memory
SAC Rd, #const Secure Address Clear	$Err \leftarrow (SR[RF[Rd]].Valid == 0) \mid$ $\quad (PC \leq SR[RF[Rd]].PClo) \mid (PC \geq SR[Rf[Rd]].PChi)$ $CCR[SX] \leftarrow Err$ $const == 0 \Rightarrow$ $\quad Mem[SR[RF[Rd].VAlo..SR[RF[Rd]].VAhi] \leftarrow 0$ $SR[RF[Rd]].PClo \leftarrow 0$ $SR[RF[Rd]].PChi \leftarrow 0$ $SR[RF[Rd]].Valid \leftarrow 0$ $SR[RF[Rd]].VAlo \leftarrow 0$ $SR[RF[Rd]].VAhi \leftarrow 0$	Verify this PC can clear the protections; if ok, do we need to clean memory to prevent leaking of sensitive information? Clear out and free the corresponding entry in SR table, regardless.
SAM* Ras, Rae, Rad Secure Address Move	$\forall\, n,\ \exists\, m \in\ (RF[Ras]..RF[Rae])$ $\quad (SR[n].valid == 1\ \&\ SR[n].VAlo \leq m \leq SR[n].VAhi))\ \Rightarrow$ $\quad Err \leftarrow Err \mid (SR[n].VAlo \neq RF[Ras]) \mid (SR[n].VAhi \neq RF[Rae])$ $CCR[SX] \leftarrow Err$ $\forall\, n,\ SR[n].VAlo == RF[Ras]\ \Rightarrow$ $\quad SR[n].VAhi \leftarrow RF[Rad] + SR[n].VAhi - SR[n].VAlo$ $\quad SR[n].VAlo \leftarrow RF[Rad]$ $\forall\, addr \in [RF[Ras]..RF[Rae]]\ \Rightarrow$ $\quad Mem[RF[Rad] + (addr - RF[Ras])] \leftarrow Mem[addr]$ $\quad Mem[addr] \leftarrow 0$	Over the start-end source range, be sure it's a perfect match, otherwise throw an exception; for every range match, fix the guard to point to the new addr block for post-move security and then set up the actual Moving protected data, erasing the source

Table 2: Security Aware instruction extensions (*SAx*) that manipulate the SR table. *PC* is the PC of the currently executing instruction. If no error is encountered, the actual instruction behavior is then executed. *RF* denotes the main register file and *Mem* is memory, while *SR* is the security register set and the *CCR* is the condition-code register with the *SX* bit corresponding to a security exception condition, causing a fault. The ∀ searching nature suggests the use of CAM cells for the given field in the SR table. *This instruction is only executable from super-user mode (i.e., typically via a kernel syscall).

bits set accordingly (i.e., any xor, add, shift, etc. that uses a "secure" register value automatically becomes "secure" flagged). Any attempt to execute a load/store operation on any register which is annotated as secure enabled requires the same authorization step – namely, passing the security check from the SR table. If this security check fails, then the contents must be encrypted prior to writing to memory.

3.2.4 Register spills, Page tables

One drawback to the enhancement in the register file is the inability for backing up and restoring secure-aware flag during register spills. This is handled by exploitation of the ECC memory bits [25]. The ECC memory bits, which are typically unused, provide approximately one extra bit per machine word. With registers marked "secure", the only way to preserve the integrity of sensitive information is to encrypt the store to memory. However, the encryption process cannot expand the machine word via padding or other methods. Therefore, we propose to use the ECC bit corresponding to each machine word as an *encrypted data* flag. When storing to memory from a secure register to an insecure location, the contents are encrypted with a key embedded in the processor. There may be several such keys used on a random basis to reduce the chance of brute-force attacks. Regardless, during write-out the ECC bit is set. When that memory location is read back, the ECC bit indicates that the contents need to be decrypted *and* that the secure bit in the register file should be set. Otherwise any load from a non-secure memory location resets the security bit in the register file for the destination register.

The exploitation of ECC bits, however, gives rise to the problem of page tables and swapping to disk. In general, the practice of swapping secure or encrypted information to disk should not be encouraged. In the specifics of our InfoShield system, we require that any page that contains secure or ECC-flagged encrypted data must be pinned in memory. Therefore, it cannot be swapped to disk to risk sensitive information disclosure or the loss of the ECC encryption bits that indicate scrambled memory contents.

By presenting an argument where encryption exists internal to the processor, a strong question is whether just encrypting all sensitive data is sufficient without our proposed architectural changes. In reality, just encrypting memory in this manner is vulnerable to a brute force attack, since the contents of memory can still be read. With the restrictions on padding, brute force attacks are possible. Our solution, however, prevents the attacker from even reading the contents of secure memory from the running process, which eliminates the brute force approach. Compromising another process and reading this processes memory is of little use since any other process will not know which memory is encrypted and which memory is not.

3.2.5 Garbage Collection and Block Copy

Many modern languages, as well as older languages with support libraries, provide garbage collection systems. Garbage collection (GC) violates the abstractions introduced here to protect sensitive memory.

By nature, GC tries to walk through all of dynamic memory to facilitate compaction and freeing of bulk heap space. However, with no *a priori* knowledge, GC systems will inadvertently attempt to access or move protected memory, causing security exceptions.

Our solution is via a privileged security-aware *move* instruction (SAM in Table 2). This instruction is capable of moving the contents of protected memory, but is unable to return the contents of protected memory or to alter the next-authorized instruction windows. Its purpose is primarily to support GC algorithms of any type. A typical GC that runs on a security aware platform would have to implement exception handlers to catch security violations (i.e., via *sigaction()* and friends). During GC data movement, any movement that causes the security exception to be returned turns into an iterative per-word movement until the protection region is found and the complete protected region size is determined. Once found, a syscall is made to the OS to request privileged movement of the sensitive data from the old to new address. This movement does alter and update the address range in the SR table, yet it does not impact the authorized PC in the application. While our solution does require minor modifications to any general GC implementation, once implemented it will work for all programs. The same technique is also applicable to secure block copy operations using library routines such as *memcpy()*.

On those architectures supporting complex memory-to-memory copy operations, such as a *string copy or "movs" instruction* in x86, the principle is similar. Such CISC instructions are actually decoded into a series of *μops* to set up a loop and execute individual load/store operations. If any *μop* load/store fails due to a security violation, the re-order buffer is tagged as receiving the security exception for the original *movs* instruction. This in turn can be caught by signal handlers which switch from *movs* to the iterative block copy routines.

3.2.6 Multiple consumers or producers

Certain coding constructions such as virtual tables, switch statements, and function pointer lookups create scenarios where there may exist multiple valid next-target PCs. Even simple code hammocks may try to access sensitive data or create sensitive container in each path, requiring some mechanism for support.

First, we support defining of regions that cover instructions which may access sensitive data. Rather than have several entries, we can define a large window of code that continually operates on sensitive data, when the sensitive information is first put under *guard* a first-next-PC and a last-next-PC may be specified. Second, we support multiple entries in the SR table for a given sensitive address. That is, if address 0xA000 is protected, multiple *SAP* entries may have this address for different consumers of the data. That we support this repetition in the secure register table is the reason the "for all"(∀) constructs exist in Table 2 security extension instructions. In particular, any instruction attempting to register *additional* next-valid PCs must be itself authorized.

3.2.7 Context switching

A potential issue for any security method that uses write-only architectural state as our SR table is handling a context switch. Since no instruction is capable of reading the contents of any SR register field, support for context switching must be moved into the necessary supporting instructions. We expect that every process has a private process space in memory for storing context information. If a unique key exists inside each system, then the hardware may write the SR table to the process space in memory encrypted with the secure key. No additional overhead or management by the OS is required. Since the addresses in the SR table are all virtual, no side effects of operations such as page table modification will impact the security features.

3.2.8 Dynamic or shared libraries

Verification of secure information usage across dynamic libraries

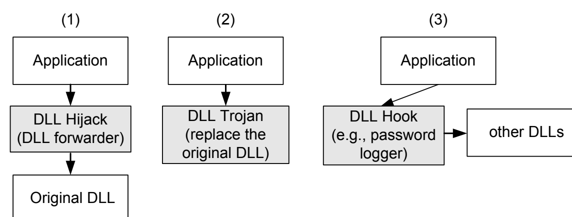

Figure 3: Trojan DLL

is not trivial, since frequently the caller has no precise knowledge of the callee code that will use sensitive information. To make things even worse, systems such as Windows use DLLs heavily for normal execution of applications. This allows third-party DLLs to be inserted or hooked into an application program that can intercept and detect messages communicated between applications and libraries, as well as between different libraries [12]. The types of information that can be intercepted include passwords, account ids, keystrokes, and so forth.

Figure 3 shows three ways of exploiting a typical Windows DLL vulnerability for information theft. First, a malicious DLL that has the same name as a legitimate DLL (such as *wsock32.dll*, the Windows system DLL responsible for network connections) can be injected into user or kernel space. The malicious DLL exports all the symbols of the original DLL and hijacks all calls to the original DLL. It intercepts sensitive information from a hijack before delegating the call to the original DLL function. For example, a worm can install a malicious *wsock32.dll* and rename the original as something else. Then each time after the system is booted, network applications such as IE will call the malicious wsock32 for transmitting passwords, username, credit card numbers, etc. The hijack DLL can examine the data before forwarding them to the original DLL. If sensitive information is found, it may mail it out to an outside email account. Second, an information stealing Trojan can replace or patch the original DLL. The new DLL contains malicious code that can disclose sensitive data to outside world. Examples of such attacks are socket Trojans such as Happy 99 [27] and Hybris [27]. Third, Windows allows arbitrary DLLs to be hooked into an application's memory space [12]. Any malicious DLL downloaded either by an Internet worm or unwillingly invited to the system as a Trojan horse can be hooked into an application's memory space. This DLL will register a callback function for handling sensitive user information. Many password stealing or keyboard input logger trojans are based on the vulnerability of Windows hooks [8].

InfoShield tackles this issue through a sequenced signature verification, which relies on a safe signing/certification of libraries with versioning. During application development, calls to external libraries are annotated with a required library version and the library public key in use at compile-time. This signature and version information is embedded in the application image. During the initialization of the *main()* sequence, all required external libraries are pre-loaded and their OS fingerprints are checked. This process involves the operating system loading and verifying the internal signature of the library, ensuring that no tampering has taken place from what the original developer provided for certification. This OS fingerprint is followed by verifying the developer signature on the library code with the applications' embedded public key.

Within any DLL that manipulates sensitive information, protection is enforced by a *policy* of secure instruction placement within the DLL function body. This policy can easily be enforced by the compilers. The premise to the policy is that as versions of a DLL change, there is no safe *a priori* method for knowing *which* instruction within a DLL function `foo` will be the first to require security permissions. Therefore, on entry any function that will manipulate sensitive data

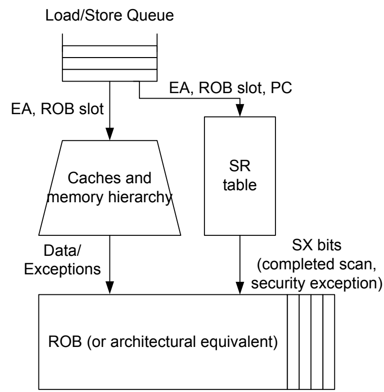

Figure 4: Decoupling SR table check and lazy update of ROB.

must execute as its first instruction the necessary setup for the secure permissions within the body of that function. Therefore, even if the first working instruction that uses sensitive data is moved to a new address during version changes, the external program is unaware and the system operates normally.

Given the condition that DLLs or other shared libraries are signed and certified, applications can seal sensitive data based on the signature/certificate. This simple solution prevents almost all variants of the three exploit scenarios using Trojan or malicious library hooks. For DLL hijacking, the internal application version signatures will fail to match resulting in aborted execution. In the case of DLL replacement, the malicious code module will have a different signature, thus it cannot properly unseal the information since each signature generates a unique key. The third scenario of DLL hooks can also be prevented in a manner similar to the case of DLL hijack.

3.2.9 Critical Path

One potential performance downside in InfoShield is caused by the fact that every load/store operation must check the SR table. This is likely to become the critical path in the memory pipeline. We propose to decouple the SR table check such that it is off the critical path entirely, with the SR check results pushed into the reorder buffer for evaluation at retirement.

Our proposed change is shown in Figure 4. As effective addresses (EAs) are moved into the memory hierarchy for access, the EA and its associated PC (along with the ROB target) are also passed to the SR table. Once the pipelined SR lookup resolves whether the access is valid, it updates the two *SX* bits in the reorder buffer to indicate that checking is complete (bit 0), and whether an exception was generated (bit 1). During retirement, any incidence of a violation in the SX bit causes a fault to be generated. This can then be handled by the OS, or passed to the user for signal handling.

4. SECURITY ASSESSMENT AND LIMITATION

InfoShield is a novel scheme for protecting usage of sensitive data against local and remote software exploits. It is aimed to counter realistic attacks on disclosing sensitive user or system information through: 1) direct or covert memory scan by malware [15]; 2) invalid input to an application or function call (CAN-2004-0415, CAN-2004-0535, GENERIC-MAP-NOMATCH) [20]; 3) buffer overrun based information theft (CAN-2004-0165)[20], or 4) malicious DLL hooks or API hijacking Trojan [8, 32, 27].

Using special dynamic protection on information usage, InfoShield can enhance the assurance that sensitive data is used, shared or disclosed in the way as defined by application semantics. InfoShield improves protection on data privacy against many documented attacks, exploits or vulnerabilities. One advantage of InfoShield is that it is compatible with the current software system model. It supports programs or libraries from different sources (sometimes un-trusted) to inter-operate and co-exist in the same memory space. More importantly, InfoShield does not require that every program component in an application's memory space must be benign, exploit-free or bug-free. However, InfoShield does not prevent all the possible attacks or exploits on data privacy, especially information theft using sophisticated physical attacks. Specifically, InfoShield protects sensitive information stored in the memory from

- theft through memory scan
- leakage through function call interception
- unwanted disclosure via hacker induced pointer/index overflow

In reality, there is no silver bullet for solving all the data privacy issues and exploits. It is preferred that a combination of many techniques such as a firewall, safe programming, or information flow analysis with InfoShield be used to prevent cyber theft. Furthermore, InfoShield assumes separate protection on program integrity. It requires that code image is properly signed with digital signature and certified. Given a program that is properly designed, InfoShield ensures that during execution there is no misuse or abnormal use of information caused by many real local or remote exploits. When a process finishes using sensitive data, InfoShield's *erase* operation resets the memory state preventing accidental leakage [3]. The OS can also ensure that memory pages allocated to a user process are properly cleared. InfoShield also assumes that applications are executed in released mode instead of debug mode. Debug mode execution often needs random access to memory that would require break of InfoShield protection. To prevent production application from being executed in debug mode, we extend the signed code image with a special debug mode disable flag. For released software, this flag is set. Since the code image is signed and certified, it is not possible to wiggle the flag without being detected.

5. EVALUATION AND ANALYSIS

To evaluate the idea of information usage protection and the specifics of InfoShield design, we used a number of network applications that manipulate sensitive user data such as a login password, cryptographic keys, or other credentials. A straightforward implementation of InfoShield can be achieved by extending the ISA and adding necessary compiler support. For evaluation purpose, we manually identify sensitive data based on the application source and annotate the application for emulation-based evaluation. We used a open-source IA32 full system emulator — Bochs [14] which models the entire platform including the network device, VGA monitor, and other devices to support the execution of a complete off-the-shelf OS and its applications. We use the memory tainting technique similar to that in [3]. Different from the objective of [3]'s study that focused on proper cleanses of sensitive information after their lifetime, we focus on the guarantee of proper usage of sensitive data during their lifetime.

We used RedHat distribution of Linux as our target system. We evaluated eight network server/client applications. They are file transfer server (wu-ftp daemon), web server (Apache http daemon), email sever (imap daemon), ftp client, a text based web browser called Lynx, an email client Pine, Openssh daemon (sshd) and Openssh based secure file transfer server (sftp). We manually annotated each application's source code such that sensitive data "live ranges" are explicitly exposed to the Bochs emulator. The implementation is similar to the memory tainting technique in [3]. In addition to memory tainting, we also applied register tainting that keeps track of instructions operating on sensitive data.

5.1 Sensitive Information and Usage Protection

For all the eight applications, we identified the global and local variables for storing the sensitive information and the program codes that operate on the information. To avoid unnecessary over-protection, we consider only the following information as critical sensitive data that requires InfoShield protection,

5.1.1 User password

All Unix and Linux applications use the well-known shadow password authentication approach. On the server side, user passwords are stored in an encrypted format or cryptographically encoded format. This is done by invoking *crypt* function with the input text set to NULL and the key set to be the password. Crypt is a one way hash function. This is an algorithm that is easy to compute in one direction, but very difficult, if not impossible, to calculate in the reverse direction. When a user picks a password, it is cryptographically encoded with crypt along with a salt value. Note that salt itself is not sensitive data. The result, i.e. shadow password, is stored in the hard disk. When a user logs in and supplies a password, the supplied password is cryptographically encoded and then compared with the encoded shadow password loaded from disk. If there is a match, then the user is authenticated. During the authentication process, user's password will reside in the server's memory space and vulnerable to memory scan, pointer overflow, or crypt function interception attacks. We can use InfoShield to ensure that during user password's lifetime, it is only used by the proper authentication routines as defined by the program semantics. Applications can declare the password's memory space as sensitive after it is first time loaded into memory.

5.1.2 Openssh host key

Openssh uses the well-known asymmetric key approach for user authentication. For each server, there is at least a pair of private and public keys. In general, only the private key is considered as sensitive and requires protection. In Openssh implementation, private keys are stored in a global data structure called "sensitive data". Openssh memsets sensitive data to zero after they are no longer needed. To protect private keys from theft during its lifetime, we declare their memory space as sensitive data and apply InfoShield protection. It ensures that only the private key authentication routine can access those keys in the way as defined by the authentication semantics.

5.1.3 AES cryptographic keys

For each data channel, Openssh uses AES standard for encryption and decryption. For each new connection, Openssh daemon will spawn a new child process. The AES keys are stored in each child process's memory space and used for network data encryption/decryption. The AES key's lifetime spans from the beginning to the end of a connection. During this time, it is vulnerable to attacks such as memory scan and invalid pointer/array index exploits. In our evaluation, we declared AES keys as sensitive information. The requirement is that they can only be used by the AES encryption/decryption routines based on the AES implementation. Any other access including access from other part of the Openssh daemon is considered as a protection violation of InfoShield.

Note that we only treat the original password and cryptographic key as sensitive information. Password digests or encrypted passwords or encrypted keys are not considered as sensitive data. In real systems, password digests or encrypted passwords are not treated as secrets and often stored in files that allow public access.

5.2 Analysis of Performance Impact

The performance impact of InfoShield on application execution is very small given that comparing with the whole application, the amount of data and code that handles passwords, cryptographic keys or other sensitive data is small. To verify this projection, we used the memory tainting technique that keeps track how sensitive data is accessed during its lifetime. The goal is to show that the overall number of memory accesses to the sensitive data and the total number of instructions directly operating on the sensitive data during any sensitive lifetime are both negligible.

In our evaluation, all the applications are tested in real-time with Bochs emulation. We used automatic scripts whenever possible or manual interaction. For OpenSSH, the test consists of connecting to the server, login with authentication, and run a list of popular shell commands. To simulate an ssh server scenario, the test included four concurrent ssh connections. For sftp, the test consists of connecting to the server with authentication, upload and download a set of files. To simulate a server setting, the test used six concurrent sftp connections. Test of wu-ftp server was similar to the sftp test case except it used wu-ftp server instead of the OpenSSH ftp server. For Apache, the Bochs emulated server hosts a website where access to the webpages requires proper user authentication. The test uses wget, a popular command line web page download tool. It automatically supplies download requests with user password and recursively downloads all the web pages from an URL. For imap server (an email server), the test consists of running a python script that automatically connects to the server as a user with password, and retrieves all the emails. For ftp client, the test is similar to the ftp server test with the difference that the client not the server is executed in the emulated platform. Both Pine and Lynx are client applications that require user interaction. Test of Pine consists of connecting to a mail server, supplying password and reading received emails. Test of Lynx consists of connecting to a web server, browsing some webpages that require user authentication. The middle column of Table 3 shows the number of profiled instructions for each application.

Applications	Instruction Count	Sensitive memory blocks required per process
OpenSSH	887803740	18
sftp	1430629493	18
httpd (Apache)	809343773	1
ftpd	575053604	1
imapd	795433147	1
ftp	564446668	1
Pine	656122506	1
Lynx	1189131107	1

Table 3: Dynamic Instruction Count and Sensitive Memory Blocks

With the Bochs emulator, we are able to measure the percentage of accesses to the sensitive information comparing with the total number of regular memory accesses. Figure 5 shows the results. For five of the eight tested applications (ftpd, imapd, ftp client, Pine, and Lynx), the percentage is less than 0.002%. For OpenSSH, the percentage is 0.1% and for sftp, 0.6%. Apache has the highest percentage of 1.8%. One of the contributing factors why Apache has so many accesses than others is that Apache verifies password for each file downloaded from a server. Typically, one webpage may contain hundreds of small files. This leads to frequent access to user password. OpenSSH applications have more frequent access due to the reasons that they use AES to encrypt/decrypt every network packet. The average is 0.3%.

Figure 6 shows the percentage of dynamic instructions that directly operate on sensitive information (load instruction included). For all the tested applications, the percentage is below 1%. For some applications, the number is below 0.001%. The average is 0.2%.

Results in Figure 5 and Figure 6 show that for typical applications, the amount of memory reads and dynamic instructions dealing with sensitive information are both very small. Since InfoShield has performance impact only on the part of application codes that handle sensitive information, its overall performance impact would be very small.

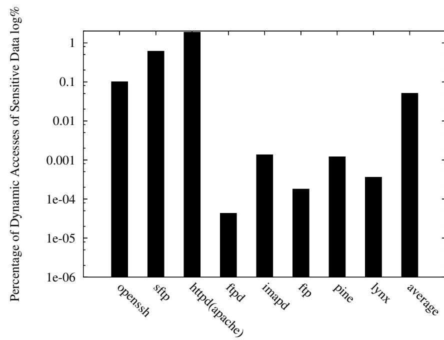

Figure 5: Percentage of accesses to passwords and keys over all memory accesses (log scale of % data).

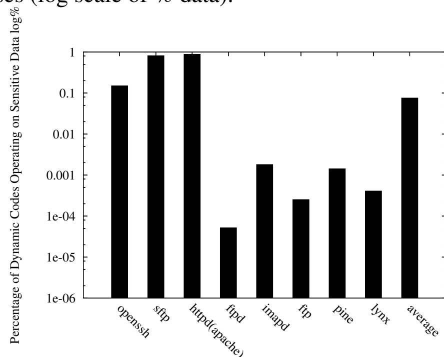

Figure 6: Percentage of dynamic instructions directly operating on sensitive information (log scale of % data).

For all the tested server applications, for each new connection, the server process will spawn a new child process for processing the new connection. The number of sensitive data regions is really small if considered at per-process basis. For shadow password authentication, the number is one (region where plaintext user password is stored). For OpenSSH, the number is six per process. Based on the test result, increasing the number of concurrent connections does not affect the number of sensitive data regions per process as each new connection is handled by a separate child process.

5.3 Sizing the Security Register Table

To determine the number of slots required in the SR Table, as outlined in Section 3.2.1, the same benchmarks were evaluated for their unique contiguous regions of memory that contain sensitive information. The result of this analysis is shown in the third column of Table 3. They suggest that the SR table itself can be kept quite small, less than 32 entries even for complex applications. These values represent the secure regions over the complete dynamic lifetime of an application, and do no consider that not all regions may be active at the same phase of execution. The small requirements for the table also result in faster access and lower power dissipation.

6. RELATED WORK

There are many approaches proposed before to address the protection of information privacy using either software or hardware mechanisms. InfoShield, however, is the first one as to our knowledge that protects dynamic usage of information according to program semantics.

6.1 Proof carrying code

Proof carrying code is a concept that users of a program can verify the safety of a program provided by an untrusted source. Using formal methods and first order logic, the program carries with itself a proof that it will abide with certain clearly-defined safety policies of the user. The proof can be verified by the user before execution. If the proof can be verified, the user can be convinced that the program is safe to be executed [22]. Proof carrying code has been extensively studied for mobile code security where producers of mobile code provide proofs and consumers of mobile code validates the proof. However, writing proof carrying code program is a daunting task for regular programmers as it requires understanding of formal logic and safety theorem proving. Another issue is that it is not clear whether the proof system is powerful enough to capture all the security constraints of a software system. In contrast, information usage safety is a much simpler concept and protection of information usage as proposed in this paper can be fully automated.

6.2 Information flow analysis

Information flow analysis is a language based technique that addresses the concern on data privacy by clarifying conditions when flow of information is safe. Through information flow analysis [7, 29], it can be enforced that high privilege and high security level information would not flow to channels with low privilege or low security level. Information flow based protection constraints unsafe flow of information. Recently, dynamic tracking of information flow for protecting data privacy using hardware approach is also proposed to confine flow of information in execution time [31]. As we have addressed before, there are many differences between the concept of information flow safety and information usage safety. Unlike information flow safety, information usage safety itself does not restrict sharing and disclosing of information as long as such operations are carried out according to program semantic.

6.3 Safe language based protection

Another language based effort to secure data privacy is strongly typed language that ensures type safe access of private information [13]. For example, a type safe language defines what operations a piece of code can perform on objects or data of a particular type. Type safety can be either enforced statically such as Java or dynamically at run-time. Information usage safety and InfoShield are different from language type safety based protection. Information usage as proposed provides more protection on information because it restricts that the protected information be only used in a specific manner while type safety only ensures that the information is operated by operations that can access the type of data.

6.4 Static language based checks

There are compiler and programming language based techniques that automatically check buffer overflow or memory reference bugs in application source code [5, 4]. Those techniques can mitigate some of the risks of disclosing sensitive data by ensuring that the applications satisfy certain safety standards. However, different from *InfoShield*, they cannot provide real-time safeguard on the access and usage of information.

6.5 Memory reference monitor and Mondrian

Hardware based memory protections such as virtual memory protection, user/supervisor execution levels, process memory space isolation and etc are based on checking the privilege of addresses issued by executing instructions to the memory. As we have addressed, these protections provide only very coarse level protection on memory and they cannot prevent exploits using malicious code residing in the same memory space as the application or exploits that induce misbehavior of the application to disclose sensitive information. Mondrian is a hardware architecture motivated to provide fine-grained access control on memory at small granularity such as word level [34]. The memory state of InfoShield is also fine-grained with each 32-bit dword having its own state. In this aspect, it is similar to Mondrian. But the similarity ends here. Mondrian is not designed to han-

dle many of the exploits or vulnerabilities on data privacy mentioned in this paper.

6.6 Hydra

Hydra [18] was one early object-oriented capability-based system which separated access control mechanism from security policy. Follow up operating system such as KeyKOS [24] also used similar idea of capability. A capability is a token that designates an object and indicates a specific set of authorized actions (such as reading or writing) on that object. Every object in the system was protected and any access required capabilities. InfoShield is very different from Hydra-like capability-based systems. First, Hydra-like systems divide everything into objects and enforce access control on every objects which lead to a system that is too general and complex. InfoShield is a simple and efficient mechanism which only aims to protect critical information in memory. Second, Hydra-like systems only care about the access control of objects, it cannot enforce proper sequences of information usage according to the program definition or semantic in real execution time.

6.7 Tamper resistant systems

Tamper resistant systems using memory encryption [19, 30] are related in the sense that tamper resistant systems also provide data privacy by storing encrypted data in memory. There are many fundamental differences between InfoShield and tamper resistant systems. First, InfoShield is designed for protecting data privacy against many software based exploits and vulnerability without using memory encryption. Overhead of InfoShield is much smaller than [19, 30]. Second, InfoShield enforces information usage safety (information used exactly the way as defined by application) while current tamper resistant systems do not have this notion. It is believed that tamper resistant systems are also vulnerable to exploits of pointer references, array indexes. Also it is not clear how tamper resistant system based software platform supports DLL hooks and tackles trojan horse injection. InfoShield solves these issues without the assumption of memory encryption.

7. CONCLUSION

This paper presents *InfoShield* that provides architectural and programming support to protect usage of sensitive information against many documented attacks and exploits on data privacy including memory scan, pointer/array index manipulation, integer overflow, format string attacks, and password-stealing trojans. By embedding specialized verification and tracking instructions inside the applications, InfoShield is capable of ensuring that secrets such as passwords and access credentials are accessed only in the way as defined by the given application. Such authenticated information usage provides a safer environment for network applications where disclosure of sensitive information due to remote or local software exploits is a major concern. Comparing with prior works on protection of data privacy, InfoShield is a much light-weight alternative and would incur much less performance penalty based on application profiling.

8. ACKNOWLEDGMENT

This work is supported in part by NSF grants CCF-0326396, CCF-0447934, CCF-0430021, CNS-0325536, and a DOE Early CAREER Award.

9. REFERENCES

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A Core Calculus of Dependency. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1999.

[2] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[3] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security Symposium*, 2004.

[4] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *USENIX Security Symposium*, 2001.

[5] 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 *USENIX Security Symposium*, 1998.

[6] CyberCrime. http://www.ssg-inc.net/cyber_crime/cyber_crime.html.

[7] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[8] B. Friesen. Passwordspy - retrieving lost passwords using windows hooks. *http://www.codeproject.com/*.

[9] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and Hash Trees for Efficient Memory Integrity Verification. In *Proceedings of the Ninth Annual Symposium on High Performance Computer Architecture*, 2003.

[10] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1998.

[11] iDefense. http://www.idefense.com/advisory/01.28.03.txt.

[12] I. Ivanov. Api hooking revealed. *http://www.codeproject.com/*.

[13] A. K. Jones and B. H. Liskov. A language extension for expressing constraints on data access. *Commun. ACM*, 21(5):358–367, 1978.

[14] K. Lawton. Welcome to the Bochs x86 PC Emulation Software Home Page. http://www.bochs.com.

[15] A. Kumar. Discovering passwords in the memory. *http://www.infosecwriters.com/text_resources/*, 2004.

[16] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *USENIX Security Symposium*, 2001.

[17] P. Laud. Semantics and Program Analysis of Computationally Secure Information Flow. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, 2001.

[18] R. Levin, E. Cohen, W. Corwin, and W. Wulf. Policy/mechanism Separation in Hydra. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 1975.

[19] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. B. J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems*, 2000.

[20] MITRE. http://cve.mitre.org/.

[21] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

[22] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119. ACM Press, 1997.

[23] G. C. Necula and P. Lee. Efficient representation and validation of proofs. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, 1998.

[24] N.Hardy. The keykos architecture. In *Operating Systems Review*, 1985.

[25] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *International Symposium on High-Performance Computer Architecture*, 2005.

[26] Scut. Exploiting format string vulnerabilities. 2001.

[27] SecurityResponse. http://securityresponse.symantec.com/.

[28] A. Shamir and N. van Someren. Playing hide and seek with stored keys.

[29] G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *Proceedings of the Symposium on Principles of Programming Languages*, 1998.

[30] E. G. Suh, D. Clarke, M. van Dijk, B. Gassend, and S.Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing . In *Proceedings of The Int'l Conference on Supercomputing*, 2003.

[31] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th International Symposium on Microarchitecture*, 2004.

[32] VirusLibarary. http://www.viruslibrary.com/.

[33] D. Volpano and G. Smith. Verifying Secrets and Relative Secrecy. In *Proceedings of the Symposium on Principles of Programming Languages*, 2000.

[34] E. Witchel, J. Cates, and K. Asanovic. Mondrian Memory Protection. In *Proceedings of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[35] www.ncipher.com.