Ensuring Critical Data Integrity via Information Flow Signatures

William Healey, Karthik Pattabiraman, Shane Ryoo, Paul Dabrowski,
Zbigniew Kalbarczyk, Ravi Iyer and Wen-Mei Hwu

Center for Reliable and High Performance Computing
Coordinated Science Laboratory

ABSTRACT

Memory corruption attacks represent one of the largest attack classes observed in the field. Many techniques have been proposed to protect applications from memory corruption attacks. However, these techniques cannot be applied selectively to security-critical data without nullifying their guarantees. We present a technique to provide protection of select critical data from a wide range of memory corruption attacks. The technique computes the backward slice of each critical data item using compiler-driven static analysis and ensures that data within the slice are not corrupted at runtime. The checking is performed using a combination of hardware and software. We evaluate the technique on three real server programs and find that the performance overhead of selectively protecting security critical data is about 10%.

Categories and Subject Descriptors

D.3.3 [**Security]: Compilers, Information-flow, Runtime checks**

General Terms

Design, Experimentation, Security, Languages, Information flow signatures.

Keywords

Critical data, backward slice, static analysis, hardware-based checks, trusted instructions.

1. INTRODUCTION

Memory corruption attacks represent one of the largest attack classes observed in the field [36]. Since languages such as C and C++ are not memory-safe, a pointer to a memory object can be manipulated by an attacker at runtime to overwrite nearby objects in memory. We define a memory corruption attack as a malicious modification of a pointer causing it to write to an object other than its referent object. This represents a violation of the intended behavior of the program, since according to the C language semantics [30], the results of a memory-unsafe operation are undefined.

The main contribution of this paper is a technique for protecting critical data in a program from a wide variety of memory corruption attacks using the concept of information-flow signatures. Examples of security-critical data include user account data in a banking application and variables that hold information about user-authentication in an SSH application. While traditionally memory corruption attacks have targeted control-data in applications (e.g. return addresses, function pointers), it has been shown that non-control data can be just as easily attacked (corrupted) to gain control over programs [2].

Using compiler-based static analysis, the technique proposed in this paper extracts the backward program slice [25] of critical program data[1] and encodes the slice in the form of a signature, referred to as the *Information-flow signature (IFS)*. The signature is enforced during application execution using a combination of software and

[1] In this paper, we use the term data to refer to both variables as well as memory objects in the program.

programmable hardware. Since the analysis is based on the properties of the program according to the language semantics, valid code is never rejected[2] by the technique i.e. the technique has no false-positives.

The proposed technique is based on the well-known observation that the main reason for a memory corruption attack is the 'gap' between the program's source-level semantics and its execution semantics. *Information flow signatures (IFS) enforce the source-level semantics of memory accesses at runtime thereby closing this gap for critical data.* The technique ensures the integrity of the protected data even if other data in the application is compromised or controlled by the attacker. As a result, the technique can be applied selectively to protect critical data in the application with significantly lower overheads compared to existing techniques for memory safety [3][12][13] which take an 'all-or-nothing' approach and are unable to provide the same guarantees when applied selectively to the program (as shown in section 7).

The IFS technique is implemented as a two-level checking scheme. The first level check needs to be applied for all instructions in the program and is therefore implemented in hardware (through processor pipeline modifications). The second level check needs to be applied only to those instructions that belong to the backward slice of critical data, which is usually a small subset of the program's instructions, and is currently implemented in software.

In order to reason about the guarantees provided by the IFS technique, we have built a formal model of the technique and have used this model to prove that the integrity of the program critical data is preserved in the face of memory corruption attacks. Attacks are represented as the modification of the destination of an instruction in a given instruction sequence to a different object (or variable) than the one specified by program semantics.

We also measured the performance overheads for runtime checking of results of information-flow signatures for three real server programs namely, SSH, FTP and NullHTTPd. In each case, a particular security-critical functionality of the application is protected. Since the technique is implemented substantially in hardware, the performance overhead of checking is constant and is about 10%. The software checks had negligible overhead in our experiments.

1.1 Threat Model

The aim of the technique is to preserve data integrity rather than its confidentiality: hence, the technique does not address side-channel attacks [15]. The threat model assumes that the attacker can execute arbitrary code and overwrite program variables stored in memory and processor registers as long as the malicious memory accesses are performed through the processor. Malicious DMA transfers are *not* covered by this threat model -- for example an attacker could use an IEEE 1394 interface port to initiate transfers to main memory of a system which are not visible to the processor [31]. We also require that the load path of the program be trusted, which can be accomplished through secure bootstrapping procedures [35]. The technique is immune to attacks on the operating system after program loading is completed (as long as the attacks do not disable the hardware checker).

Examples of attacks covered in the threat model include:

- Classical memory corruption attacks such as buffer overflow, format string, and heap corruption attacks that overwrite non-control data in the application. These attacks violate the source-level semantics of the program and are caught by the technique. We assume that other techniques such as control-flow integrity [6] or program shepherding [1] can be deployed to protect control-data in the application.
- Insider attacks, in which the attacker attempts to alter (at runtime) part of the program to gain control over the critical data. An example of this class of attacks is a malicious plugin for a web-browser that tries to modify sensitive data in the browser in violation of its interface with the browser. The attack will be detected even if the plugin's code is unavailable at compile-time as the checks are in the browser's source code.

[2] Some applications may write past the end of an object into another memory object during correct execution. We do not consider such programs.

2. APPROACH

Memory corruption attacks are performed by causing an instruction to write to an address outside the bounds of the instruction's valid destination object. Therefore they can be prevented by checking the bounds of every write to memory. However, performing bounds checks on every write at runtime is prohibitively expensive. Static analysis techniques have been proposed to eliminate certain runtime checks [12][13]. However in languages such as C/C++, it is very hard to statically determine the targets of memory reads and writes and eliminate checks. Even with type-safe languages, one still needs to perform some checking during runtime as attacks are performed by distorting the behavior of instructions during program execution.

We observe that checking every write is excessive when a user is only interested in protecting certain critical data. The goal of our technique is to check a minimal number of instructions while ensuring that the critical data is not corrupted. *Since we are applying protection selectively, it is insufficient to check only the direct writes to critical data. This is because while protecting direct writes makes it difficult for the attacker, a smart attacker can still influence the value of a critical data indirectly as shown below:*

Suppose program variable *c* is calculated by adding two other variables, *a* and *b*. Assume that *c* is part of the critical data in this application. An attacker can influence the value of *c* indirectly by corrupting either *a* or *b*. Thus, in order to protect critical data from corruption, we ensure that all data and instructions that the critical data is dependent upon are also protected. This constitutes the backward program slice [25] of the critical data. In general, the backward slice contains the set of instructions and data objects that both directly and indirectly influence the critical data.

The protection scheme consists of two phases: A) a compile-time phase to extract the backward slice of critical data in the program, and B) a runtime-phase to check if the critical data is influenced in violation of the statically derived backward slice. The runtime phase is implemented using a combination of software and hardware.

A. Compile-time Phase, a compiler-based static program analysis determines the following:

1) The instructions that can influence the critical data (according to program semantics)

2) For each instruction in (1), the set of objects (data) that the instruction is allowed to write to

The compiler marks each instruction in (1) and each object in (2) as *trusted*. This *trusted* property is propagated at runtime according to the propagation rules defined in Section 4.2.

B. Runtime Phase, The following invariants are enforced by a combination of hardware and software at runtime.

1) Critical data is modified only by *trusted* instructions and objects (enforced in hardware)

2) Each *trusted* instruction writes only to its statically allowed objects (enforced in software)

If either property is violated, an interrupt is triggered which halts execution of the program and raises a security alert *before* critical data is corrupted. This is vital to ensure fast application recovery, from a checkpoint for example (we do not consider recovery techniques further in this paper).

3. STATIC PROGRAM ANALYSIS

This section describes the approach to derive information flow signatures for critical data in applications. Without loss of generality, we refer to critical data as critical variables in this section, although in reality critical data encompasses both memory objects and program variables. The steps in the compiler analysis are as follows:

1) *Determining the security-critical variables* for the program. This is done by the programmer using source code annotations, based on an understanding of the program semantics (for example, variables used in authenticating a user in an SSH server).

2) *Constructing the backward program slice.* The compiler then constructs the backward program slice for each of the critical variables. This slice contains all the data and instructions that legitimately (according to source-code semantics) influence the critical variable either directly or indirectly in a program execution. The

backward slice is computed inter-procedurally in the program (the implications of this are discussed in section 3.1).

3) *Encoding the dependencies.* After extracting the backward slice, the compiler stores information identifying the critical variables and their extracted dependencies (both instructions and data) either in a separate configuration file, or by embedding it in the executable. This information is configured into the hardware module that performs runtime checking at program load time.

3.1 Data-flow and Pointer Analysis

In order to extract the backward slice of the security critical variables, the compiler performs pointer and data-flow analysis on the program. While data-flow analysis typically involves data dependences through registers and pointer analysis tracks data dependences through memory. Data-flow analysis answers the question of which instructions can directly or indirectly affect certain registers, while pointer analysis answers the question of which instructions can potentially write to specific memory variables [32]. This is done by deriving *points-to* sets of pointer variables used by the store instructions. The points-to sets of the loads and stores are then used to construct a conservative superset of the data flow arcs between these instructions. While register data-flow analysis is a standard technique in most compilers and is highly accurate, pointer analysis is a much harder problem and is typically addressed with various degrees of approximation depending on the space/time tradeoffs [14].

The approximations made by the compiler generally over-estimate the memory data-flow graph, thus affecting the checking resolution of the derived signatures and possibly allowing attacks that could have been avoided in a more accurate analysis. Appendix A discusses the effects of using imprecise pointer analysis on the security of the IFS technique.

In order to ensure that the protection is as high as possible, we use a precise pointer analysis scheme (context-, field-, array- and heap-sensitive analysis according to the classification in [14]) to compute the points-to sets of instructions in the program. The effect of flow-insensitivity is considered in the next section.

3.2 Effect of Flow-insensitivity

A flow-sensitive analysis is one which considers the order of updates to a pointer variable within a function. For example, consider the piece of code in Table 1a. In the code pointer *p* initially points to the address of variable *x.* It is then updated with the address of variable *y,* and is finally assigned to the pointer *q.* A flow-sensitive analysis will reason (accurately) that pointer *q* can only point to the variable *y*. This is shown in Table 1b. A flow-insensitive analysis on the other hand, will assume that both *p* and *q* can point to either *x* or *y,* as it does not consider the order in which the statements are executed. This is shown in Table 1c.

Table 1: Example showing the effect of flow-insensitive analysis on security

a)	***b)***	***c)***
$p = \&x$;	$p \leftarrow \{ x \}$	$p \leftarrow \{ x, y \}$
$p = \&y$;	$p \leftarrow \{ y \}$	$p \leftarrow \{ x, y \}$
$q = p$;	$p \leftarrow \{ y \}, q \leftarrow \{ y \}$	$p \leftarrow \{ x, y \}$ $q \leftarrow \{ x , y\}$

A smart attacker could take advantage of the imprecision introduced by the flow-insensitivity and make either pointer *p* or *q* point to the address of object *x.* This can be accomplished by changing the control-flow of the program, or by overwriting the value of the pointer *p* (or *q)* through a memory error. In order to achieve useful results with the attack, the variable *y* must belong to the backward slice of critical data in the program (so that the attacker can influence the critical variable). This problem can be avoided if within a single function, the same pointer is not used for accessing both variables that belong to the backward slice of critical data as well as variables that do not. Our compiler analysis performs limited single-static assignment (SSA) transformation to achieve partial flow-sensitivity.

3.3 Discussion

In this section, we consider some limitations of our current compiler analysis and how discuss this affects the coverage of the proposed technique.

- We do not protect control-data dependences in the program. For example, if variable *A* is used as part of a branch decision in computing the value of a second variable *B*, *B* is control-data-dependent on *A* [32]. In our current implementation control-data-dependences are not included in the backward slice of signatures. As a result certain attacks may escape detection, as shown in section 5.2.
- For some applications, it may not be possible to encode the entire backward slice starting from the critical variable and going back to the beginning of the program, as the slice may be too large to fit in the hardware module that performs runtime checking. While this was not the case in the applications considered in Section 6, in general it may be necessary to constrain the number of levels up to which the backward slice is encoded by the technique. *We believe that encoding the slice for even a few levels is often sufficient in practice. This is because in order to be undetected by the technique, an attacker must corrupt data prior to the encoding of the slice in the program. Moreover, it may be difficult for an attacker to cause data corruption that propagates to the critical variable without crashing the program or resulting in failed assertions.*

4. RUNTIME CHECKING

This section presents a two-level checking scheme that enforces the information flow signature and detects attacks memory corruption attacks before the critical data is corrupted. The two levels are summarized below:

- The first level ensures that only certain instructions (called trusted instructions) can write to the critical data.
- The second level ensures that trusted instructions only write to the trusted data they are allowed to write to by the source semantics (as determined by the static program analysis in section 3).

4.1 Terms and Definitions

Before presenting the checking scheme, we introduce relevant terminology:

Critical Data: A variable or object, which if corrupted by an attacker, could lead to the security of the application being compromised. The determination of critical data can be done either automatically (through heuristics) or manually (using knowledge of program semantics as done in this paper)

Trusted Instruction: An instruction that influences critical data directly or indirectly through dependencies, as determined by static analysis. All the instructions in the backward program slice of the *critical data* are considered trusted instructions.

Trusted Data: A variable or object that influences the value of *critical data* (through a *trusted instruction*).

4.2 Level 1 Check: Trustedness

The objective of the Level 1 check is to separate the critical or trusted data from the non-critical or un-trusted data (according to static analysis). *In the context of the information flow signature, Level 1 ensures that instructions outside the backward slice of a critical data item (i.e. un-trusted instructions) do not influence instructions inside the signature (i.e. trusted instructions).*

The Level 1 check is implemented by maintaining a trusted bit (in hardware) for each register and memory location in the program. The initialization and propagation of the trusted bit is performed by hardware in parallel with instruction execution according to the rules described below.

Trust Propagation. Trust is propagated during an instruction's execution according to the rule shown below. *I* stands for the current instruction and its fields are shown as *I.dest, I.pc* and *I.op1 …opN*.

Trused(I.dest) ← Trusted(I.operands) && Trusted(I.pc)

where:

Trusted(I.operands) = Trusted(I.op1) && Trusted(I.op2) …. & Trusted(I.opN)

The destination register or memory location (*I.dest*) is marked trusted if and only if all the instruction's operands (*I.op1 to I.opN*) are marked trusted and the instruction itself (*I.pc*) is marked trusted. If any one of these conditions does not hold, the trustedness of the destination is cleared. The runtime tracking of the trusted bit is similar to the tracking of the Taintedness bit in [16], with the difference that the tainted bit is set if ANY of its operands are tainted. The details of the hardware scheme for trusted bit propagation are presented in section 4.4.

Detection: An alarm is raised if a trusted instruction attempts to use an un-trusted data operand OR if an attempt is made to write to a critical data item through an un-trusted instruction. The first clause protects against instructions that attempt to indirectly influence the critical data, in violation of statically derived source code semantics. This is because, under attack-free execution, all operands of a trusted instruction would be marked trusted. If an operand to a trusted instruction is un-trusted, it means that the operand was influenced by an un-trusted instruction or un-trusted data, and hence should not be allowed to influence the critical data. This protects the critical data from indirect modification by untrusted instructions. The second clause protects the critical data from direct modification by untrusted instructions.

The detailed exposition of the above rules under various conditions is shown in Table 1.

Destination / (I.pc, I.operands)	Critical	Non-Critical
(Trusted, Trusted)	Pass to Level 2	Pass to Level 2, set trusted bit of target
(Trusted, Un-Trusted)	Raise Alarm	Raise Alarm
(Un-Trusted, Trusted)	Raise Alarm	Allow, clear trusted bit of target
(Un-Trusted, Un-Trusted)	Raise Alarm	Allow, clear trusted bit of target

Table 2: Conditions and corresponding runtime actions for Level 1 check

4.3 Level 2 Check: Verification of Trust

The Level 2 check enforces two invariants: (1) Trusted instructions cannot influence a critical data item to whose backward slice they do not belong. (2) Instructions inside a backward slice can only write to critical data items that are *directly* dependent upon them inside the backward slice.

Note that the Level 1 check needs to be performed on all instructions, both trusted and un-trusted. In contrast, the Level 2 check is performed only on trusted instructions that pass the Level 1 check. In the current implementation (discussed in this paper), the Level 1 check is performed in hardware while the Level 2 check is performed in software. This allows the Level 1 check to be performed very efficiently for the vast majority of the program's instructions, while the slower Level 2 check needs to be performed only for relatively few instructions.

Data Structures: In order to facilitate the Level 2 check, two tables are maintained by the runtime system.

- *Data Address Range Table (DART),* which maps virtual memory addresses to the corresponding variable or memory object identified by the compiler. The mechanism for keeping track of the address ranges of various types of memory objects is described below.
- *Instruction to Allowed Data Table (IADT),* which maps each trusted instruction to the data that are the allowed targets of the instruction as determined by the compiler. This table is initialized at compile-time and remains unchanged during program execution.

Level 2 Checking Mechanism:

1) Upon execution of a trusted instruction, the checking mechanism retrieves both the address of the instruction (stored in the program counter) and the destination address to which the instruction is attempting to write.

2) The destination address is used to index the *DART* to determine which variable or object the runtime instruction is actually attempting to modify.
3) The program counter is used to index the *IADT* to determine which objects or variables the current instruction is allowed to modify.
4) The data items identified in steps 2 and 3 are compared with each other to ensure the destination of the instruction is within in its allowed-write set. If this is not the case, an alarm is raised and the program is halted (the write is not allowed to complete, thereby preserving the memory state before the write).

Runtime Mapping of Data. In order to determine the memory object or variable for a given virtual address, the runtime mechanism must have the ability to map virtual addresses to the corresponding symbolic name used by the compiler. This information is maintained in the *DART* depending on the type of object as follows:

- *Global Variables, Static Variables or Constants*: These variables are statically allocated at compile time and hence their addresses are known and can be stored in the DART at program load-time.
- *Local or Stack Variables:* The exact address of these variables is not known statically or at program load-time, but can be represented statically as an offset on the stack with respect to the base pointer of the current function. These are also stored into the DART at program load-time.
- *Dynamically allocated Heap Objects:* The address mappings of these objects are not known statically, but are determined at runtime by intercepting calls to the memory allocator. This requires instrumentation of *malloc* and *free* library calls, in order to store the address returned by the *malloc* into the DART.

Protection of Data Structures: In order to ensure that the attacker cannot overwrite the data structures used in the scheme, the DART can be protected either by using the operating system's page-level access controls, or by applying the technique to itself with the DART marked as critical data.

4.4 Hardware Implementation

The technique is implemented using a combination of hardware and software. The hardware implementation requires the addition of a special CHK instruction to the instruction set of the processor, but no other direct modifications to the pipeline. The processor's caches are not modified. Instead a content addressable memory (CAM) outside of the processor's pipeline is used to store information about the trusted and critical bit of instructions and data. The Level 1 check is implemented in hardware, which guarantees the check has a low performance overhead and that it is performed on every executed instruction. The Level 2 check described in Section 4.3 is implemented in software since it is only executed on trusted instructions, which consist of a small subset of the total program as noted in section 3.2. We describe the hardware implementation in this section.

A prototype is implemented in Field Programmable Gate Array (FPGA) hardware within the Gaisler Research Leon 3 open-source VHDL processor [20]. The hardware module which implements Level 1 checks snoops required signals from the Leon 3 processor pipeline through a generic interface to the pipeline. These signals include: 1) register file control, 2) current instruction and its pointer, 3) pipeline stall, flush, and 4) cache control. Signals are used directly from the processor's pipeline without modifications. The checking module is shown in Figure 1. It contains a pipelined structure similar to the main processor's pipeline that is used to compute the trusted bits assigned to each data item in the program. Signals read from the processor pipeline are used to control this checking pipeline. Outputs from the checking module trigger an interrupt within the processor, allowing software to handle Level 2 checks and security violations.

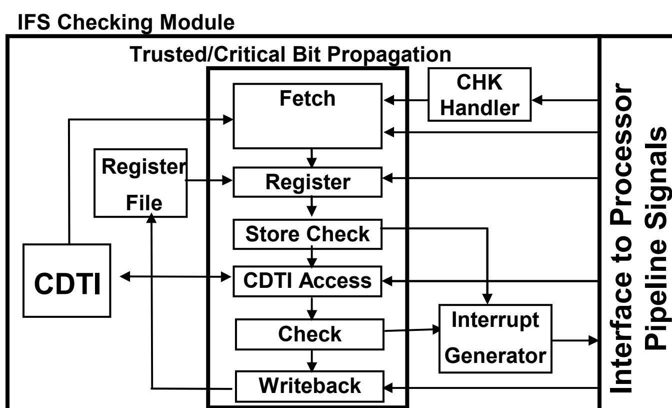

Figure 1: Schematic of hardware checking module

Table 3: CDTI - Critical and Trusted Bit Storage

Address	Instruction	Critical
0x000012c0	1	0
0x00041400	0	1
0x00041404	0	0
…	…	…

Initialization and System Assumptions: High-level communication used to initialize the checking module with the trustedness bits and the critical bits occurs through CHK instructions. The information is loaded from an instrumented program at load time by using these CHK instruction – recall that the program load path is trusted, as described in Section 1.1. CHK instructions are interpreted as no-operations by the main processor, and thus to be used only by the checking module as it snoops on signals from the main processor. In this current implementation, we assume that only one program utilizes the technique at a given time. Future work will focus on extending the hardware mechanism to allow multiple applications to use the technique simultaneously. Although the implementation of the checking module is specific to the Leon 3 processor, it is possible to implement the technique for other single-issue processors as long as the pipeline signals mentioned above are available. Future work will examine requirements to implement the technique on superscalar processors.

Storage of Criticality and Trustedness Information: A CAM embedded within the checking module is used to hold the addresses of trusted and critical data and trusted instructions in the program. It is referred to as the CDTI, for Critical Data/Trusted Instruction. Likewise, a register file within the module contains single bit registers which indicate whether the corresponding registers of the main processor hold critical data.

The use of the CDTI to store the trusted and critical bits obviates the need to use system RAM to mark every address's criticality and trustedness. This technique also relieves the need to add an extra bit to the main bus width within the main processor, or tag caches with extra information. Such approaches, which have been used in [21] and [22] for different processor-level security enhancements, require significant effort to modify and re-validate the design of the processor and call for changing architectural characteristics, such as bus widths, of current systems.

The structure of the CDTI is shown in Table 3. The first column contains the addressees of variables and instructions marked trusted. The second column signifies the corresponding entry is an instruction, rather than a variable. The third column shows if the corresponding entry is for critical data.

Operation: The operation of the checking module in Figure 1 is as follows:

- During program initialization, CHK instructions read from the main processor pipeline enter the *CHK handler* within the module and are used to initialize the CDTI.
- During runtime, the *fetch stage* looks up each instruction's trustedness within the CDTI based on the program counter value read from the main processor pipeline.
- Trusted instructions have their operands looked up in the *register stage* of the module.
- The *store check* stage of module enforces Level 1 checking rules for store instructions, before they enter the memory stage of the processor. (e.g. if a trusted store instruction uses non-critical operands, the checking module raises an alarm before the memory operation occurs)
- *CDTI access* looks up the trusted and critical bit variable information in the CDTI using cache control signals from the processor.
- In the *check* stage, trusted instruction operands are checked and the destination of untrusted instructions is checked and actions are taken as shown in Table 2.
- In the *writeback stage*, trusted bit information is propagated back to the register file.

4.5 Formal Model

We have built a formal model of the IFS technique and use the model to prove that the integrity of critical data is ensured by the technique when the program experiences memory corruption attacks. *Given a sequence of instructions and the critical data that needs to be protected, we show that the system never enters a compromised state (where a compromised state is defined as one in which the critical data is allowed to be influenced by an untrusted instruction).* Attacks are represented as the modification of the destination of an instruction in a given instruction sequence to a different object (or variable) than the one specified by program semantics.

The system model is similar to the Bell and LaPadula model [26]. The model and proof are presented in Appendix B. The following assumptions are made in the proof:

1) *An attack does not change one valid control path into another valid control path*. We do not consider control-flow attacks as described in the threat model (Section 1.1).
2) *Instructions cannot be modified during execution*. This can be enforced through page-level protection since we do not consider self-modifying programs.
3) *Each instruction in the backward slice of a critical variable writes to a single object*. In some programs, this can be accomplished by the compiler through selective inlining and partial loop unrolling if necessary in the backward slice. Figure 2 below shows why this assumption is necessary for the proof and how programs that violate this assumption are handled.
4) *The compiler knows statically which object each instruction in the backward slice of a critical variable is allowed to write to according to source-code semantics*.

It is important to note that these assumptions are restrictive in order to model the technique for all possible attacks. For example, the IFS scheme provides protection for certain programs even if they violate assumptions 3 or 4. For other programs, such as those that contain address calculations based on user input (as shown in Figure 2), a compiler warning is generated with the location of the vulnerability.

```
void chrcpy(char *dst, char*src) {
 *dst = *src;
}

main() {
 char bufA[10];
 char bufB[10]; //Critical
 int Aindex;

 input Aindex;

 chrcpy(&bufA[Aindex], &bufA[0]);
 chrcpy(&bufB[1], &bufB[0]);
}
```

Figure 2: Example of a program that violates assumption 3

Figure 2 shows an example program for which assumption 3 would be violated. In the program, the *chrcpy* function copies the character pointed to by the second argument to the memory pointed to by the first argument. Suppose that the programmer has determined that *bufB* is critical. Since both *bufA* and *bufB* are used as inputs to the *chrcpy* function, the set of valid destination objects of the **dst = *src* instruction includes both *bufA* and *bufB*. By supplying a value of 10-19 for *Aindex*, an attacker can cause the *chrcpy* function to write to *bufB* when it should be writing to *bufA* (during the first call to chrcpy). The Level2 check cannot detect this attack since both *bufA* and *bufB* are valid destination objects of the **dst = *src* instruction. This attack can be prevented by inlining the *chrcpy* function such that one copy of the **dst = *src* instruction writes only to *bufA* and the second **dst = *src* instruction writes only to *bufB* (and thereby satisfying assumption 3).

Notice that in the example above the attacker has a valid path through which he/she can influence the value of a pointer (through the *Aindex* offset), and thereby determine to which the data object it writes. It is important to

note that these attacks would not be possible if the program did not contain a valid path through which user data can influence a pointer address.

5. PUTTING IT ALL TOGETHER: SSH Example

The authentication function in the OpenSSH server program [27] is used to demonstrate the two-level checking scheme described in Section 4. This is one of the three applications used to demonstrate the information flow signature technique in Section 6. Due to space constraints, the other two applications, namely WuFTP [28] and NullHTTPd [29] considered in section 6 are not described at this level of detail.

Figure 3 shows a code snippet from the *SSH* server program that is used to authenticate a user based on the user supplied login and password. In this case, the *encrypted_password* variable is the password entered by the user and the *pw_passwd* variable is the encrypted password that the system reads from the password file. The goal is to protect the return value of the function, the *authenticated* variable, which we mark as critical, as an attacker can illegally authenticate themselves by overwriting the *authenticated* variable.

```
int sys_auth_passwd(Authctxt *authctxt, const char *password) {

        1:  struct passwd *pw = authctxt->pw;  char *encrypted_password;
          /* Just use the supplied fake password if authctxt is invalid */
        2: char *pw_password = authctxt->valid ? shadow_pw(pw) :  pw->pw_passwd;
        /* Check for users with no password. */
        3:  if (strcmp(pw_password, "") == 0 &&   strcmp(password, "") == 0) return (1);
        /* Encrypt the candidate password using the proper salt. */
        4:    encrypted_password = xcrypt(password, (pw_password[0] && pw_password[1]) ? pw_password : xx");
        /* Authentication is accepted if encrypted passwords match */
        5:    int authenticated = (strcmp(encrypted_password, pw_password) == 0);
        6:    return authenticated; /*Critical Data*/
}
```

Figure 3: Example code fragment from SSH showing how the runtime checks protect the critical data

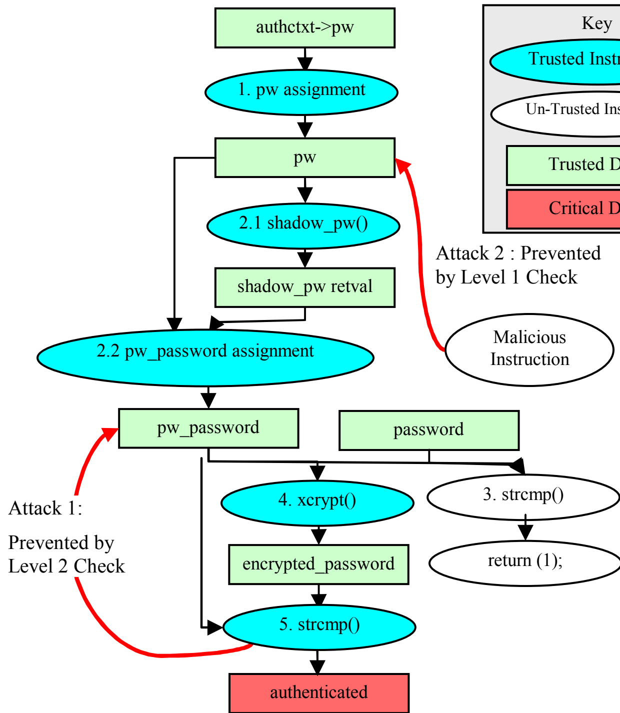

Table 4: Mapping of trusted instructions to objects

Trusted Instruction	Data objects/variable allowed to be modified by trusted instruction
*1. assignment of (*pw)*	pw
2.1 call to shadow_pw()	shadow_pw retval
*2.2assignment of (*pw_password)*	pw_password
4. call to xcrypt()	encrypted_password
5. call to strcmp()	authenticated

Figure 4: Static Dependence Graph corresponding to the SSH example code

For simplicity, the statements within the function body are annotated with integer labels that correspond to instructions in the slightly simplified version of the compiler-derived static dependence graph for the code shown in Figure 3. Table 4 shows the mapping between the trusted instructions and the data-objects/variable that they are allowed to modify (based on inter-procedural pointer analysis performed by the compiler). This corresponds to the IADT (Instructions to Allowed Data Table) described in Section 4.3.

We show through examples how the technique is able to detect arbitrary attacks, regardless of how the attacker attempts to corrupt the critical data. We do not make assumptions about whether there exists a valid input path leading to the critical data corruption. The attacks are also indicated in Figure 4.

5.1 Attack 1 : Attacker overwrites *pw_passwd* variable

Assume that the *strcmp* statement at Statement 5 has a memory error and allows the attacker to overwrite the *pw_passwd* variable and influence the results of the comparison (presumably authenticating the attacker with an incorrect password). Since instruction 5 is marked trusted, it is checked by the Level 2 check, which will lookup instruction 5 in the trusted instruction table (shown in Table 4). According to the table, the only object or variable that can be modified by instruction 5 is *authenticated*. Since the instruction has instead attempted to write to *pw_passwd*, the attack will be detected by the Level 2 check.

5.2 Attack 2 : Attacker changes the *pw pointer* variable

Assume the attacker tries to overwrite the value of the pointer to *pw*, possibly to cause *shadow_pw* to return a hash for which the password is known. The attacker must modify *pw* from an un-trusted instruction, since all trusted instructions in the function are checked by the Level 2 check. However, when the pointer to *pw* is written by an un-trusted instruction, its trusted bit is cleared as per Table 2. Now, when the call to the *strcmp* function (which is a trusted instruction) is executed, it attempts to use the *pw_password* variable. Since the trusted bit is cleared for this variable, an alarm will be raised by the Level 1 check and the attack will be detected.

5.3 Attack 3 : Attacker changes value of authctxt->valid

Since the *authctxt->valid* variable is used to decide whether to obtain the shadow password, an attacker could change the value of *authctxt->valid* from 0 to 1 and force the system to obtain the shadow password even if an

invalid username is supplied. This may be used by an attacker to gain access to the system using expired or revoked accounts for which the password is known. This attack would not be detected by the IFS scheme since modifying the *authctxt->valid* field makes the program execute a valid (but incorrect) control-path. The reason for this vulnerability is that the IFS technique does not consider control-data dependencies as part of the signature. (see Section 3.3). Adding control-dependences to the signature would provide detection for this attack.

6. EXPERIMENTAL EVALUATION

This section describes the experimental evaluation of the performance overheads of information flow checking. The technique is evaluated using three open-source server programs, namely OpenSSH, WuFTP and NullHTTPd. We first describe the three applications, the hardware evaluation methodology and then the software evaluation methodology and results.

6.1 Programs Description

The three programs used to evaluate the technique are described below. For OpenSSH and WuFTP, we created a stub version that mirrors the functionality of the original program with respect to the security critical function being studied. The original program source lines that are retained in the stub are not modified in any way.

The creation of a stub allows us to focus on the important parts of the application, without losing the critical functionality being studied. When constructing the stubs, we used natural program boundaries to determine what functions to include in the stub. For example, the SSH stub contains all the functionality of the keyboard-interactive authentication mechanism. The stubs are representative of truncating the backward slice of critical data in the programs (as described in Section 3.3).

OpenSSH: OpenSSH is a freely available implementation of a Secure Shell (SSH) Server in C [27]. We focus on the functions that authenticate the user based on the user-supplied password. The original application consists of 66278 lines of C code and the stub version consists of 224 lines of C code.

We chose the following variables as critical data for the application, as their corruption can result in incorrectly authenticating an attacker.

1) *authctxt:* a pointer to the authentication context, which includes the user entered password.
2) *result*: the return value of the *sys_auth_passwd()* authentication function, the value of which determines whether the user is authenticated.
3) *fakepw*: a pointer to a dummy authentication context used when the user has entered an invalid username.
4) *permit_empty_passwd*: a flag which determines if users are allowed to login using empty passwords

WuFTP: This is a freely available open-source implementation of a File Transfer Protocol (FTP) server written in C [28]. We focus on the functions which report the actions of the user to the system log. Maintaining an audit trail is a security-critical functionality as an attacker may attempt to cover his/her tracks after an attack. The original application consists of about 23534 lines of code and our stub consists of 523 lines of C code.

In WuFTP, the critical data includes:

1) *syslog:* The data buffer containing the string logged by FTP.
2) *checkauth:* Indicates whether a user is successfully authenticated in the system .
3) *path:* Holds the path where a transferred file will be stored by FTP.

NullHTTPd: NullHTTPd is a small and efficient multithreaded HTTP server for both Linux and Windows operating systems. For this program, we protect the entire program, rather than extracting a stub version. The application consists of 2300 lines of C code.

In NullHTTPd, our aim is to protect the webserver from memory corruption and spoofing attacks. The critical data identified includes:

1) *char *file:* A pointer to the buffer which holds the webpage which is to be sent to a client.

2) *mime_types*: Holds the various MIME types that the webserver can output, and their associated extensions.

6.2 Software Evaluation

The *IMPACT* compiler [17] is used for static analysis and *IMPACT's Lemulate* tool is used to implement the Level 2 checks. *Lemulate* allows IMPACT to transform a 3-address intermediate representation to a C-code representation rather than a program binary. This C-code file can then be instrumented, re-compiled into a binary, and executed. The Level 2 checks are implemented in software as a small C runtime library (Section 4.3).

The Level 2 security checks are automatically inserted as calls to the checking library before each trusted instruction in the C-code generated by *Lemulate*. The function calls include the following information as arguments: (i) the ID of the instruction (corresponding to the PC which would be used in a hardware implementation), (ii) the virtual address of the object the instruction is writing to, and (iii) the size of the object the instruction writes to. For static objects, these values correspond either to compile-time constants or to simple arithmetic expressions involving the stack pointer. For dynamic objects, the compiler identifies the *malloc* call that creates the object (allocation site), extracts the variable that contains the address and size of the object at the allocation site and passes it to the checking library function.

For the applications we studied, the security critical functionality (captured in the stub) was not on the performance critical path due to natural modularity and encapsulation in these programs. Therefore, the checking overhead was dominated by the Level 1 check in hardware (shown in Section 6.3). Since the performance overheads for the Level 2 check are low, we report the number of checked instructions (by Level 2) and local/dynamic variable mappings for each application in Table 5. This corresponds to the sizes of the DART and IADT structures described in Section 4.3. Note that in the cases of OpenSSH and WuFTP, the number of checked instructions is identical for the stub and for the entire program since the stub represents the truncation of the backward slice within the full program as mentioned in Section 6.1.

Table 5: Number of instructions that need to be checked in the applications considered

	Critical Data	Total Ins. in Program	Total Ins. in Stub	Checked Ins.	Local Variable Mappings	Dynamic Object Mappings
OpenSSH	*authctxt*	42435	500	125	84	3
	result			126	84	3
	fakepw			4	3	0
	flag			16	11	0
WuFTP	*auth*	23060	810	7	12	0
	syslog			28	22	0
	path			45	29	0
NullHTTPd	*file*	7095	7095	62	31	0
	mime			136	56	66

OpenSSH

The total percentage of checked instructions in SSH for all critical variables is less than 1%. The following observations can be made from the table for the OpenSSH application:

- The *authctxt* and *result* variables share the same 125 instructions and 84 local variables. This overlap between the dependence trees of these two critical variables is a significant advantage. For example, if the *authctxt* pointer variable is already being checked, the *result* variable can also be checked by adding a single trusted instruction. Hence, the incremental overhead in checking both variables is extremely low.
- Another interesting result is the low number of instructions required for checking the *fakepw* variable. This variable points to a dummy authentication context which SSH hashes to authenticate against in the event that the provided username is invalid, or the account is either locked or disabled. If an attacker is able to

overwrite the hashed value associated with this *dummy user* with a hash for which he/she knows the password, the attacker would be authenticated in the system. Since very few legitimate instructions use this dummy authentication context, it is extremely efficient to check, and arguably prevents an important security attack. To the best of our knowledge, such an attack is hitherto undiscovered.

WuFTP

As can be seen from the table, the percentage of checked instructions in FTP is less than 1% of the total instructions in the program, even when the total number of checked instructions for all critical variables is considered. The reasons for the relatively fewer number of checked instructions in WuFTP are as follows:

The authentication mechanism of FTP is much simpler than the authentication mechanism of SSH. This is because FTP does not use encryption or complex handshaking in its authentication. Therefore, there are very few instructions in the backward slice of the critical variable *auth*.

NullHTTPd

The percentage of checked instructions for NullHTTPd is less than 2% percent for the *file* and *mime* variables, resulting in low checking overhead. We also considered the protection of the *conn* variable inside NullHTTPd which is a pointer to a linked list element containing information about the current connection. However, this variable turned out to be a poor choice of critical variable due to its large backward slice containing 1486 instruction and 975 variables. This is because *conn* is passed around and used in a large percentage of the functions in the program. Therefore, care must be exercised in the choice of critical data, to balance the performance degradation with the security provided.

6.3 Hardware Evaluation

The Leon 3 open-source processor has a 7-stage pipeline and has been configured to include a 4-way set associative 32-KB data cache and 4-way 8KB instruction cache. The rest of the system on a chip is a DDR controller and Ethernet core. The prototype hardware design is synthesized for a Xilinx Virtex-II Pro 30 FPGA using Synplify Pro v8.1, with place-and-route procedures completed by Xilinx ISE 8.2.

The minimum size of the CDTI for implementing the Level 1 checks for an application is given by the maximum of the sums of the last three columns in Table 5. Of the applications considered, *WuFTPd* required a CDTI consisting of at least 128 entries, while the *OpenSSH* and *NullHTTPd* applications required a CDTI of at least 256 entries each.

Performance: The original clock frequency target for the system is 65 MHz. With the checking module with a 128-entry CDTI added to the processor, the resulting clock frequency is 64.3 MHz. This corresponds to a performance overhead of about 1%. A 256-entry CDTI corresponds to a performance overhead of about 10%, due to the routing constraints within the FPGA. These constraints would be alleviated in an ASIC design, where signal connections are not limited to pre-placed paths as in a FPGA.

Area: The checking module with a 128-entry CDTI increases processor the gate count by 4.4%, or 109,004 gates over the Leon 3 processor's original 2,454,472. The 256-entry CDTI yields an increase in the gate count of 8.6%. This corresponds to less than 850 gates per CDTI entry, which is an extremely small footprint relative to over a billion transistors per die produced by modern semiconductor processes.

7. RELATED WORK

Much of the earlier work on preventing **memory corruption attacks** has targeted specific kinds of attacks. For example, techniques such as *StackGuard* [3] and *Libsafe* [5] protect specifically against buffer overflow vulnerabilities. Similarly, techniques such as *Program Shepherding* [1] and *Control-flow Integrity* [6] protect against attacks in which the attacker corrupts control data such as function pointers and return addresses. Young and McHugh [19] have shown that it is possible for an attacker to overwrite non-control data and take control of or change the execution of the application. Chen et al. [2] have shown that such non-control data attacks are

practical for many real world applications such as FTP and SSH. In contrast, the technique proposed in this paper does not target a particular type of attack, but limits the writes to those allowed by the original source code semantics, thereby protecting against a wide range of memory corruption attacks.

Another class of techniques, broadly called **information flow-based security**, enforces an externally specified policy on the program [18]. The policy can enforce either the integrity or confidentiality of critical data in the system. Techniques for enforcing critical data integrity classify program data as high-security and low-security and ensure that low-security data cannot influence high-security data in the program. These techniques require the user to provide the classification of a large number of variables by hand. Furthermore, this is a compile-time technique and can result in the rejection of valid code although the policy may never be violated at runtime.

A special case of information-flow based security is **taintedness checking**, which marks all externally supplied data (through user-input) as low-security and ensures that these cannot influence high-security data in the program such as pointers [7] and return addresses [16]. The advantage of these techniques is that the tainted data can be determined automatically by the compiler or runtime system. However, taintedness techniques are more effective at identifying vulnerabilities than detecting attacks as an application can use tainted data even when it is not under attack (for example, when computing an index into an array from user input). Therefore, these techniques can have false-positives when used for attack detection.

Address space randomization techniques [8][9][10][11] attempt to obfuscate the details of the underlying memory layout from the attacker rather than detecting and preventing attacks. These techniques incur relatively low overhead while preventing most current attacks. It has been argued that randomization may be broken by repeated undetected attacks on the system [23], or through program information leaks such as pointers exposed to the user. Unlike randomization based protection, the IFS technique does not require the program to be free of information leaks, or require that the details of the program layout be obscured from the attacker.

A broad class of techniques for ensuring **memory safety** of C and C++ programs has been proposed in the literature [12][13]. There are three main problems with this class of techniques: (i) In order to maintain reasonable performance overheads, these techniques perform various approximations or aggregations, which may be manipulated by clever attackers to their advantage. For example, [13] groups together all objects of the same size into a single memory pool and does not distinguish among objects in the same pool. (ii) These techniques require access to the entire application code and do not provide the same guarantees in the presence of untrusted third-party code and libraries (or require annotations written by the programmer) and (iii) Despite the protection provided by the techniques, the program may still be open to attacks that can change its control-flow and bypass the checks. Such attacks may be mounted through the unprotected parts of the application code if any exist.

Data-flow integrity guarantees memory safety in the presence of malicious attacks [3]. The main idea is to compute the data flow graph of the program at compile-time and enforce the computed data dependences at runtime. The data-flow integrity technique checks every load of a memory location to ensure it was written by the correct store instruction. However, by the time a memory corruption is detected, the data may have already been corrupted in memory, and hence the only recovery option is to restart the application. The IFS technique on the other hand, can detect the attack *before* the critical data is corrupted. This allows the critical data to be checkpointed and the application to be recovered from a safe checkpoint. Further, data-flow integrity requires that all loads and stores in the program be protected, but the cost of protecting all loads and stores is of the order of 45% to 100%. In order to reduce this performance overhead, the authors of [3] propose skipping checks on some variables but leave open the question of how to remove checks in a safe and secure manner.

Approaches such as NT-Swift [33] and Samurai [34] protect **critical data** in applications from memory corruption errors. However, such approaches are limited to accidental corruption of data due to software bugs, and do not offer protection from malicious attackers. In particular, Samurai [34] provides probabilistic protection to minimize the chances of correlated memory corruptions in the program, but an attacker can break the probabilistic protection through targeted attacks.

Memsherlock [24] is a technique for **diagnosing security vulnerabilities** based on an attack pattern or input. Similar to our technique, they compute the backward slice of specific variables that are likely causes of the vulnerability and dynamically track the dependences of these variables. However, the IFS technique differs from Memsherlock in the following aspects: (1) IFS is a detection technique rather than a diagnosis technique and hence needs to be deployed when the application is run in production settings. (2) The IFS technique does not start with an attack input or attack vector and hence needs to distinguish attack inputs from valid, legitimate inputs.

8. CONCLUSION

We have shown that Information Flow Signature Checking is a powerful technique, providing detection for a broad class of memory corruption attacks. The technique is highly configurable, allowing the user to determine the desired level of protection (depth of the signature), as well as which data to protect. A compile-time static analysis is employed to extract a backward slice which contains all instructions that directly or indirectly influence the security critical program data. The backward slice is then converted to a signature that is enforced by two levels of checks performed using a combination of hardware and software during runtime. Any violation of the pre-computed signature raises an alarm before the critical data is corrupted. The approach is employed and demonstrated in the context of three real server programs, and a prototype hardware implementation is described. For the applications considered, the overall performance overhead of runtime checking was found to be less than 10% and was constant due to hardware implementation. The software checking overhead was found to be insignificant considering application execution time as a whole for the majority of critical data identified.

Future work will involve implementing the Level 2 check in hardware and extending the compiler analysis to include control-dependences in the backward slice.

Reference

[1] Kiriansky, V., Bruening, D., and Amarasinghe, S. Secure execution via program shepherding. In Proc. of the 11th USENIX Security Symposium (Aug. 2002).

[2] S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. Iyer. Non-control-hijacking attacks are realistic threats. In USENIX Security, 2005.

[3] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In Symposium on Operating System Design and Implementation (OSDI), Seattle, WA, Nov. 2006.

[4] Cowan, C., Pu, C., Maier, D., Hinton, H., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Seventh USENIX Security Conference (San Antonio, TX, Jan. 1998).*

[5] Baratloo, A., Singh, N., and Tsai, T. Transparent fun-time defense against stack smashing attacks. In Proceedings of the 2000 USENIX Technical Conference (San Diego, CA, June 2000).

[6] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow Integrity: Principles, implementations, and applications. *In Proc. ACM Computer and Communications Security, Nov. 2005.*

[7] Chen, S.; Pattabiraman, K., Kalbarczyk, K. and Yer, R.K. Formal Reasoning of Various Categories of Widely Exploited Security Vulnerabilities Using Pointer Taintedness Semantics, *Security and Privacy in Information-Processing Systems, Book chapter, pages 82-99, 2004, Springer-Verlag.*

[8] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In Proceedings of the Sixth Workshop on Hot Topics in Operating Systems (1997).

[9] J. Xu, Z. Kalbarczyk, and R. Iyer. Transparent runtime randomization for security. In A. Fantechi, editor, Proc. 22nd Symp. on Reliable Distributed Systems --SRDS pages 260--9. IEEE Computer Society, Oct. 2003.

[10] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In V. Paxson, editor, Proc. 12th USENIX Sec. Symp., pages 105--20. USENIX, Aug. 2003.

[11] Berger, E. D. and Zorn, B. G.. DieHard: probabilistic memory safety for unsafe languages. In Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06). ACM Press, pp. 158-168, 2006.

[12] Necula, G. C., McPeak, S., and Weimer, W. 2002. CCured: type-safe retrofitting of legacy code. *In Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Portland, Oregon, January 16 - 18, 2002). POPL '02.* ACM Press, New York, NY, 128-139.
[13] Dhurjati, D., Kowshik, S., and Adve, V., SAFECode: enforcing alias analysis for weakly typed languages. *In Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06).* ACM Press, PP. 144-157.
[14] Hind, M. and Pioli, A. 2000. Which pointer analysis should I use?. In Proceedings of the 2000 ACM SIGSOFT international Symposium on Software Testing and Analysis (Portland, Oregon, United States, August 21 - 24, 2000). M. J. Harold, Ed. ISSTA '00. ACM Press, NY, 113-123.
[15] Boneh, D., DeMillo, R. A., & Lipton, R. J. On the Importance of Eliminating Errors in Cryptographic Computations Journal of Cryptology: The Journal of the International Association for Cryptologic Research, vol. 14, pp. 101-119, 2001.
[16] G. Suh, J. Lee, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. 11th International Conference on Architectural Support for Programming Languages and Operating Systems. Boston, Massachusetts. October 2004.
[17] UIUC Open-IMPACT Effort. The OpenIMPACT IA-64 Compiler. http://gelato.uiuc.edu
[18] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL), San Antonio, TX, USA, January 1999.
[19] W. Young and J. McHugh, Coding for a believable specification to implementation mapping, In Proceedings of IEEE Symposium on Security and Privacy, 2005, p. 142.
[20] Jiri Gaisler, Gaisler Research. Leon 3 Synthesizable Processor. http://www.gaisler.com
[21] Michael Dalton, Hari Kannan, Christos Kozyrakis, Raksha: A Flexible Information Flow Architecture for Software Security, in Proc. of the ACM Intl. Symp. on Computer Architecture, June 9–13, 2007, San Diego.
[22] G. Suh, C W. O'Donnell, I. Sachdev, and S. Devadas, Design and Implementation of the AEGIS Single Chip Secure Processor Using Physical Random Function, in Proc. of the ACM Intl. Symposium on Comp. Architecture, 2005.
[23] H. Page, M. Pfaff, B. Goh, E.J. Modadugu, N. and Boneh, D. On the Effectiveness of Address-Space Randomization Shacham ,Proceedings of the 11th ACM conference on Computer and communications security,pp 298-307, 2004
[24] E. Sezer, P. Ning, C. Kil and J.Xu, MemSherlock: An Automated Debugger for Memory Corruption Vulnerabilities, to appear in the Proceedings of 14th ACM Conference on Computer and Communication Security, Nov 2007.
[25] Mark Weiser. Program slicing. In Proceedings of the 5th International Conference on Software Engineering, pages 439-449. IEEE Computer Society Press, 1981.
[26] Bell, D. Elliott and LaPadula, Leonard J. (1973). "Secure Computer Systems: Mathematical Foundations". MITRE Corporation
[27] OpenSSH Server, http://www.openssh.com/
[28] WuFTP Server, http://www.wu-ftpd.org/
[29] NullHTTPd Webserver, http://nullwebmail.sourceforge.net/httpd
[30] Brian W. Kernighan, Dennis Ritchie: The C Programming Language Prentice-Hall 1978
[31] Adam Boileau. Hit by a Bus: Physical Access Attacks with Firewire. Presented at Ruxcon 2k6, 2006.
[32] Muchnick, S. S. 1997 Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc.
[33] Liang, D., Chung, P. E., Huang, Y., Kintala, C., Lee, W., Tsai, T. K., and Wang, C. 2004. NT-SwiFT: software implemented fault tolerance on Windows NT. J. Syst. Softw. 71, 1-2 (Apr. 2004), 127-141.
[34] Pattabiraman, K., Grover, V., Zorn, B.G., Samurai : Protecting Critical Data in Unsafe Languages, Technical Report (127), Microsoft Research, Aug 2006.
[35] J. Dyer, M. Lindemann, R. Perez, R. Sailer, S.W. Smith, L. van Doorn, S. Weingart, "Building the IBM 4758 Secure Coprocessor", IEEE Computer, Oct 2001, Vol 34 pp. 57-66
[36] CERT http://www.cert.org

Appendix A: Pointer Analysis Taxonomy

The main factors that decide the precision of a pointer analysis are context-sensitivity, flow-sensitivity, field-sensitivity, array-element sensitivity and field-sensitivity. This classification is based on [14].

We consider the effects of each of these analyses on security in the following subsections. Although some of the analyses are typically combined together in optimizing compilers, we will consider the effect of each kind of analyses independent of the others.

A.1 Advantages of a Context Sensitive Analysis

A context-insensitive analysis can create artificial dependencies among program variables that did not exist in the original program. This would allow the attacker to fake an artificial dependence and attack the system. Consider the following code sample

/* Caller function */	/* Callee function */
void goo() {	void foo(int* a, int* b) {
int c = 8, d = 12;	int *p = a;
int e = 15; f = 11;	int *q = b;
foo(&c, &d);	*p = *q;
foo(&e, &f);	}
}	

Function *foo* initializes the second argument (*b*) to the value of the first argument (*a*) through pointer manipulation. At the end of the function *goo*, the value of c is 12 and the value of e is 11. A context-sensitive analysis would be able to reason about this correctly.

A context-insensitive analysis, since it does not consider the calling context, would reason that the parameter *a* can take values *{c, e}* and the parameter *b* can take values *{d, f}*. Hence, it would assume that *both c and e* can assume the values of *d and f* and conclude that c can be either 12 or 11, which is also the case with *e.* Thus it has created an artificial dependence between *c* and *f*, and between *e* and *d*.

From the security point of view, this allows the attacker to influence the value of the critical data through the artificial dependence. Assume that the variable *c* was critical in the above program, and could only be assigned the value of *d* (one could imagine *d* was the computed hash value of the system password, and c is being assigned to it). Similarly *e* is critical and can be assigned only to the value of *f* (assume that *f* has the computed hash of the user password, which is stored in *e*). An attacker could exploit a memory error in the application and copy the value of *f* to *c* (ie. the attacker copies the computed hash of the user password to the variable that should hold the hash of the system password, thereby ensuring that the attacker's incorrect password is accepted). This would not be detected by the signature as the dependence *c*←*f* is a valid one as determined by a context-insensitive analysis. The problem is exacerbated in the case of library functions, which may be called on both critical data and non-critical data arguments. This allows the non-critical data to overwrite the critical data.

A.2 Advantages of Flow-Sensitive Analysis

A flow-insensitive analysis does not consider the exact order of statements in a procedure and assumes that any statement in the procedure that can write to the memory location is allowed to do so regardless of the actual control-flow in the procedure. Consider a code sample in which the program updates a variable at a specific location depending on whether a certain control-flow path was followed. In the *encrypt_password* function, the password is encrypted with a call to the *xcrypt* function only if it was a valid one.

```
void encrypt_password( ) {
        if (authctxt->valid) {
                authctxt->password = xcrypt(user_password);
                return true;
        }
        return false;
}
```

In the example above, a flow-sensitive analysis would be able to deduce that *authctxt->password* was modified only within the then clause of the if-statement ie. when *authctxt->valid* is non-zero. However, a flow-insensitive analysis would assume that the value of *authctxt->password* could be modified anywhere within the function. Consider the case when an invalid password is supplied by the attacker, and hence *authctxt->valid* is zero. The branch in the if-statement would fall through to code that would presumably not modify *authctxt->password*. However, an attacker could exploit a memory error in this code and make it modify *authctxt->password* as if the password had been considered valid (possibly authenticating themselves). This is because the flow-insensitive analysis does not consider control flow in computing the points-to set of an instruction.

A.3 Advantages of Field-Sensitive Analysis

Field-sensitivity allows disambiguation of pointers to individual fields of a *struct.* This is most useful when some fields of a *struct* are critical but others are not. Also, it may be the case that some members of the *struct* are written in a procedure but other elements are not. Using field-sensitive pointer analysis to derive signatures prevents attackers from overwriting fields of a *struct* by using instructions that write to other fields of the same *struct*. Note that this is less useful when all fields of a *struct* are critical, as by overwriting one critical field, the attacker has already compromised the system.

A.4 Advantages of Array-element Sensitive Analysis

In the case of array-element sensitivity, the security benefits are not clear. It is unlikely to be the case that some elements of an array are critical while others are not. Most times, we are only interested in determining whether a pointer can write to an array as a whole (typically a string buffer), so that the instruction that does the write is considered in the signature of the array. This would prevent the instruction from overflowing the contents of the array and writing to another critical array, but would not prevent the instruction from writing to another element of the array than the one for which it was intended. We do not believe this is likely to be a serious security risk.

One case where it may make a difference is if the application was performing allocations from the array, in effect treating the array as a memory pool from which smaller objects may be allocated. In this case, it may become necessary to track each element of the array individually.

A.5 Advantages of Heap-object Modeling

Typically, pointer analyses treat all objects allocated at the same site in the program as aliased to each other, i.e. an instruction that writes to a memory object allocated at a certain allocation site in the program, can write to any other object allocated at the same site in the program. This has interesting implications for security. There are two cases:

Case 1: Only objects of a certain type (linked list node) are allocated at a call site

void list_insert(int data) {
    Nodeptr ptr = make_new_node();
    ptr->data = data;
    ptr->next = start;
}
	Nodeptr make_new_node() {
	    Nodeptr ptr;
	    ptr = (Nodeptr)malloc(sizeof(Node));
	    return ptr;
	}

In the above example, all objects of the linked list node type are allocated in the function *make_new_node*. The function *list_insert* calls the *make_new_node* function and initializes the data elements for the pointer returned from it. Assume that the analysis is context-sensitive and hence can disambiguate one call to *make_new_node* from another. Since the pointer analysis treats all objects allocated at the same allocation site as identical, it will assume that the statement that assigns the value of *data* to *ptr->data* can write to any object allocated by the *malloc* call in the *make_new_node* function. Hence, the statement would appear in the signature of any linked list node's data element. This would be acceptable if all elements of the linked list were considered critical. However, if some nodes were critical and some were not, then it is possible for the attacker to mount an attack on the critical nodes by overflowing from an instruction that writes to non-critical nodes. Such an attack could be prevented by allocating the critical nodes at a different call site, so that the analysis can resolve accesses to these nodes uniquely.

Case 2: Objects of different types are allocated at the same call site

void list_insert(int data) {
    Nodeptr ptr = make_new_node();
    ptr->data = data;
    ptr->next = start;
}
	void* make_new_object() {
	    void* ptr;
	    ptr = malloc(sizeof(Node));
	    return ptr;
	}

Consider the same piece of code, but with the modification that all objects in the program, including linked list nodes, are allocated in a generic function *make_new_object* at the same allocation site. In this case, the pointer analysis would assume that any heap object in the program can be aliased to any other heap object (since both would be allocated at the same allocation site), and hence an instruction that writes to one heap object can write to any other heap object. Thus the instruction that assigns data to *ptr->data* can potentially write to any heap object, and appears in the signature of every heap object. This allows the attacker to corrupt any heap object from an instruction that writes to a heap object, be they critical or non-critical. This constitutes a serious security loophole.

Appendix B: Mathematical Model and Proof

This section describes the mathematical model and proof of the IFS security technique. It shows how *trustedness* is propagated, and how the propagation rules that are enforced by the hardware ensure that the system never reaches a compromised state.

Assumptions:

1) *An attack does not change one valid control path into another valid control path*. We do not consider control-flow attacks as described in the threat model (Section 1.1).
2) *Instructions cannot be modified during execution*. This can be enforced through page-level protection since we do not consider self-modifying programs.
3) *Each instruction in the backward slice of a critical variable writes to a single object*. In some programs, this can be accomplished by the compiler through selective inlining and partial loop unrolling if necessary in the backward slice. Figure 2 below shows why this assumption is necessary for the proof and how programs that violate this assumption are handled).
4) *The compiler knows statically which object each instruction in the backward slice of a critical variable is allowed to write to according to source-code semantics*.

Given these assumptions, we present a mathematical model of computation and use it to prove that the technique protects critical data memory corruption attacks. For the purposes of the model, we give the attacker the ability to modify/subvert the destination object of any instruction in the program. We assert that all the common memory corruption attacks can be modeled in this way including buffer overflows, format string attacks, and heap corruption attacks. For example, in a buffer overflow attack, the attacker causes an instruction to write past the instruction's legitimate destination object into a different object.

We first define the set notation used to model the program, and the present an inductive proof showing that as long as the rules of the Level 1 and 2 checks are enforced, it is impossible to reach a compromised state. We define a compromised state as a state in which data is marked both Critical and Un-Trusted. At an abstract level, the Trusted property means that data has not been influenced by a memory corruption error. Therefore, if Critical data is marked Un-Trusted it means that the critical data may have been influenced by a memory corruption error, and thus represents compromised state.

Category	Set	Elements	Semantics
Primitive Sets	S	$s \in \{S_1, S_2, \dots, S_n\}$	*subjects*; Instructions
	O	$o \in \{O_1, O_2, \dots, O_n\}$	*objects*; program data and variables in the form of symbolic objects output by the compiler
	R	$r \in (S \times O_n \times O)$	*requests*; Tuple of Instruction to be executed, the source objects, and the destination object
Decisions	T	t ϵ {Trusted, Un-trusted}	For instructions, the Trusted classification means the instruction is in the backward slice of the critical data. For data, the Trusted classification means that the data has not been influenced by a memory error.
	C	c ϵ {Critical, Non-critical}	The critical classification means that the data was deemed critical to the program by the programmer.

	B	$b \in (T \times T \times C)$	A tuple of: (trustedness of instruction, **minimum** trustedness of source objects/operand, criticality of destination)
Decisions	J	$j \in$ {Allowed, Denied}	Determines whether the instruction is allowed or denied based on the B of the instruction
	D	$d \in (T \times J)$	Pair of T: (Trustedness of destination, whether or not instruction is allowed to be executed)
States	V	$v \in (F \times A)$	*classification state*; All object's Trusted and Critical State
	W	$w \in R \times D \times V$	*system state*; State of the CPU (Request, Decision, All Objects' Trusted and Critical State)
Sequences	t	$t \in \{0, 1, 2, \dots\}$	*time sequence*; one instruction is executed per time unit
	Z	$z \in W^T$	*state sequence*; Corresponds to sequence of states in a program
	X	$x \in R^T$	*request sequence*; Corresponds to sequence of instructions in a program
Data Structures maintained by technique	F	$f \in B^R$, $f = (f_S, f_{TO}, f_{CO})$ where $f_{TS} \in T^S$ $f_{TO} \in T^O$ $f_{CO} \in C^O$	Mapping of Subject, Src Objects, Dest Object→ B (This represents the information contained in the CDTI of the hardware IE which instructions and objects are currently marked Trusted and/or Critical) This is explained in Section 4.4.
	M	$(m_{i,j})_{\|T\| \times \|T \times C\|}$ with $m_{i,j} \in D$	*result matrix*; maps F(Trustedness of Subject, Minimum trustedness of Src Objects, Criticality of Dest. Object)→Decision (This corresponds to matrix M shown below, and also Table 2 which describes the Level 1 check in section 4.2)
	A	$A = O^S$ where $A(s) = o_D$ for $(s, o_s, o_D) = X(n)$ for all n	Access Map of which Subjects have write-access to which objects (This corresponds to Table A shown below, and the Level 2 check presented in section 4.3)

Note: α^β is defined as all functions from the set β to the set α

Matrix M: Maps F (the tuple of Subject, Src Object, Dest. Object) to a Decision

$f_{TS}(S)$ / $(f_{TO}(O_S), f_{CO}(O_D))$	**Un-Trusted**	**Trusted**
(Trusted, Critical)	(Un-Trusted, *Denied*)	(Trusted, Allowed)
(Trusted, Non-Critical)	(Un-Trusted, Allowed)	(Trusted, Allowed)
(Un-Trusted, Critical)	(Un-Trusted, *Denied*)	(Un-Trusted, *Denied*)
(Un-Trusted, Non-Critical)	(Un-Trusted, Allowed)	(Un-Trusted, *Denied*)

Table A: Mapping of which Subjects have write-access to which objects

Subjects	Objects
$Instr_1$	Obj_B
$Instr_2$	Obj_b

Definitions:

Initial State: The **initial state** is defined as $Z(0) = (X(0), D^0, (f^0, a^0))$

where, $f^0 = (f^0{}_{TS}, f^0{}_{TO}, f^0{}_{CO})$ consisting of:

An Object (Data) is initially marked Critical only when defined critical by the programmer:

$f^0{}_{CO}(O)$ = Critical when O is defined to be critical by the programmer

A Subject (or Instruction) is initially marked Trusted when it is statically determined to either write to critical data, or write to data that is marked trusted:

$f^0{}_{TS}(S)$ = Trusted when $\exists\ X(t) = (S^t, O^t{}_S, O^t{}_D)$ for some t = 0, 1, …| $S^t = S$ and ($f_{CO}(O^t{}_D)$ = Critical or $f_{TO}(O^t{}_D)$ = Trusted)

An Object (Data) is initially marked Trusted when that object is marked critical, or when there exists a valid static instruction that uses the object as an operand and writes to data that is marked either Critical or Trusted:

$f^0{}_{TO}(O)$ = Trusted when $\exists\ X(t) = (S^t, O^t{}_S, O^t{}_D)$ for some t = 0, 1, … | $O^t{}_S = O$ and ($f_{CO}(O^t{}_D)$ = Critical or $f_{TO}(O^t{}_D)$ = Trusted)

General State: In general, a state is defined as follows: $Z(t) = (X(t), D^t, (f^t, a^t))$ for $t > 0$

Compromised State: Given a Z(t), $\exists o \in O$ | $f^t{}_{TO}(o)$ = Un-trusted and $f^t{}_{CO}(o)$ = Critical

i.e. There exists an object that is marked both Critical and Un-Trusted at a time step t

Secure State: A Z(t) for which the compromised state conditions do not hold.

Symbolic version of the algorithm which is enforced by the technique (presented in section ?):

Let Z be a sequence of states where $Z(t) = (X(t), D^t, (f^t, a^t))$ for $t > 0$, with $(S^t, O^t{}_S, O^t{}_D) = X(t)$ The hardware ensures that the following invariants hold in every state:

1. The mappings in Table A, and the trustedness of the subjects do not change:

$a^t = a^{t-1}$ and $f^t{}_{TS} = f^{t-1}{}_{TS}$,

If the destination object is in the allowed-write set of the Subject (according to table A), then a decision is produced by performing a lookup in matrix M. Otherwise, the decision is set to (Un-Trusted, Denied):

(II) If $a^{t-1}(S^t) = O^t{}_D$, $D^t = (D^t{}_1, D^t{}_2) = m[f^{t-1}{}_{TS}(S^t), (f^{t-1}{}_{TO}(O^t{}_S), f^{t-1}{}_{CO}(O^t{}_D))]$, otherwise D^t = (Un-Trusted, Denied)

If the lookup results in an Allowed decision, then the instruction is committed and the destination object's trustedness is propagated accordingly:

(III) If $D^t{}_2$ = Allowed, then $f^t{}_{TO}(O^t{}_D) = D^t{}_1$,

If the lookup results in a Denied decision, the subject/instruction is not allowed to complete execution (IE commit its results), and trustedness is not propagated. (In practice, an alarm is raised))

Theorem: *Starting from a state Z_0 (with f^0 as defined previously for some X'), and applying transition rules as given by the above algorithm, it is never possible to reach a compromised state Z(t) , where $X(t) \neq X'(t)$ for some or all $t > 0$ such that for* $(S'^t, O'^t_S, O'^t_D) = X'(t)$, $S^t = S'^t$ and $O^t_S = O'^t_S$ and $O^t_D \neq O'^t_D$.

We prove the theorem using mathematical induction on the state sequence Z.

Base Case: Given a secure initial state Z(0), the state Z(1) reached using a single transition step is secure.

Let $Z(0) = (X(0), D^0, (f^0, a^0))$be a secure state.

1) *If the destination is non-critical, then it is impossible to reach an insecure state through the execution of this instruction since critical data is not being written to. Thus, the trustedness of the destination is set according to the results of matrix M:*

If $f^0_{CO}(O^1_D)$ = Non-Critical, update of $f^1_{TO}(O^1_D)$ is permitted as defined in (III).

2) *If the destination object is critical and the instruction is trusted then:*

If $f^0_{CO}(O^1_D)$ = Critical and $f^0_{TS}(S^1)$ = Trusted, then

i) The source operands must be trusted, by definition of the initial state and either:

$f^0_{TO}(O^1_S)$ = Trusted (by definition of f^0) and either

(1) The destination object is not in the allowed-write set of the instruction according to Table A (meaning the attacker has subverted the destination object of the instruction) resulting in a Denied decision, or:

$a(S^1) \neq O^1_D$, and then D^1_2 = Denied; the operation is not permitted, or

(2) The destination object is in the allowed-write set of the instruction according to Table A, and thus the operation is permitted.

$a(S^1) = O^1_D$, and an updated of $f^1_{TO}(O^1_D)$ = Trusted is permitted.

3) *If the destination object is critical and the instruction is un-trusted, then the attacker has subverted the destination object of the instruction (since all instructions writing to critical data should be marked Trusted according to the initial state f^0). According to matrix M, this operation is denied regardless of the Trustedness of source operands.*

$f^0_{CO}(O^1_D)$ = Critical and $f^0_{TS}(S^1)$ = Un-Trusted then D^1_2 = Denied, according to Matrix M.

Thus, there exists no object $o \in O$ | $f^1_{TO}(o)$ = Un-trusted and $f^1_{CO}(o)$ = Critical, and Z(1) is a secure state.

Induction Case: If Z(t-1) is a secure state, then the state Z(t) reached by a single transition step is secure.

1) *If the destination is non-critical, then it is impossible to reach an insecure state through the execution of this instruction since critical data is not being written to. Thus, the trustedness of the destination is set according to the results of matrix M:*

If $f^{t-1}_{CO}(O^t_D)$ = Non-Critical, update of $f^t_{TO}(O^t_D)$ is permitted as defined in (III).

2) *If the destination object is critical and the instruction is trusted and the source operands are trusted then:*

If $f^{t-1}_{CO}(O^t_D)$ = Critical and $f^{t-1}_{TS}(S^t)$ = Trusted and $f^{t-1}_{TO}(O^t_S)$ = Trusted, then either

i) The destination object is not in the allowed-write set of the instruction according to Table A (meaning the attacker has subverted the destination object of the instruction) resulting in a Denied decision, or:

$a(S^t) \neq O^t_D$, and then D^t_2 = Denied; the operation is not permitted since O^t_D is not the expected destination object, OR

ii) The destination object is in the allowed-write set of the instruction according to Table A, and thus the operation is permitted.

$a(S^t) = O^t_D$, and $f^t_{TO}(O^t_D)$ = Trusted, since there exists no $(S^u, O^u_S, O^u_D) = Z(u)$ for u = 1, …, t-1 where $O^u_D = O^t_S$ and D^t_1 = Un-Trusted.

3) *If the destination object is critical and the instruction is trusted and at least one operand is un-trusted then the operation is denied, since at some previous time, atleast one of the source operands was written to in a way that resulted in an Un-Trusted decision. This means that one of the source operands was written or influenced by an instruction not in its backward slice, and thus could be corrupted:*

If $f^{t-1}_{CO}(O^t_D)$ = Critical and $f^{t-1}_{TS}(S^t)$ = Trusted and $f^{t-1}_{TO}(O^t_S)$ = Un-Trusted, then

D^t_2 = Denied; the operation is not permitted since there exists a $(S^u, O^u_S, O^u_D) = Z(u)$ for u = 1, …, t-1 where $O^u_D = O^t_S$ and D^t_1 = Un-Trusted.

4) If the destination object is critical and the instruction is un-trusted, then the attacker has subverted the destination object of the instruction (since all instructions writing to critical data should be marked Trusted according to the initial state f^0). According to matrix M, this operation is denied regardless of the Trustedness of source operands.

$f^0_{CO}(O^1_D)$ = Critical and $f^0_{TS}(S^1)$ = Un-Trusted then D^1_2 = Denied, according to Matrix M.

Thus, there exists no object o ϵ O | $f^t_{TO}(o)$ = Un-trusted and $f^t_{CO}(o)$ = Critical, and Z(t) is a secure state.

Both the base and induction cases have been proved. Hence, the result is proved.