Dynamic Tracking of Information Flow Signatures for Security Checking

**William Healey, Karthik Pattabiraman, Shane Ryoo,
Ravi Iyer and Wen-Mei Hwu**

Center for Reliable and High Performance Computing

Coordinated Science Laboratory

Abstract

Memory-based attacks represent one of the largest attack classes in the field. Many techniques have been proposed to protect applications from certain classes of memory exploits, however, few of these techniques can protect the application from all memory attacks, and few permit the protection of only select variables. We present a technique to provide protection of select variables from a wide range of memory attacks. The protection is provided by computing the dependence tree of each critical variable, ensuring that no variable or instruction within the dependence tree is corrupted by utilizing hardware supported runtime checks. We evaluate the technique using software based emulation.

1. Introduction

This paper presents a technique (information flow signature checking) to protect data that is critical to the application from a memory corruption attack. We define a memory corruption attack as a malicious corruption of any control/non-control data (in the heap, stack, or registers) through a memory error in the program. These memory errors can occur in programs written in languages such as C and C++ which are not type-safe, and hence the runtime environment allows any pointer to write to any location in memory, regardless of type. While attackers have exploited memory errors to overwrite control data such as return addresses and function pointers [1], they can also overwrite security-critical non-control data (such as passwords) by exploiting memory errors. Chen et al. [2] showed that such attacks are practical for a broad class of applications, including large server programs. Our goal is to protect application data that is critical from the security point of view, be it control data or non-control data, from memory-corruption attacks.

The strength of our technique is that it considers a very broad threat model. It is assumed that the attacker can execute arbitrary code as well as overwrite any program variable stored in memory or registers, provided the modification is observable at execution time. The threat model also covers physical attacks in which code is injected via some malicious hardware device, such

as a corrupt smart-card, as long as the memory is manipulated through the checked processor's pipeline.

Many existing techniques [3] break down when they are applied to select variables, as not protecting even one variable in the dependence tree of a critical variable allows an attacker to influence that critical variable. Our technique on the other hand, explicitly protects the entire dependence tree of the critical variable. This allows the overhead of checking to be configurable based on application requirements. Furthermore, information flow signature checking ensures that any security violation affecting the critical data can be detected before the data is used by the original program in a security-critical context. This allows attacks to be detected before they can compromise the system or propagate to other parts of the system.

The proposed technique is primarily based on the observation that *the main reason for a data error is the 'disconnect' between the source-level semantics of a program and its system semantics*. Information flow checks enforce the source-level semantics of memory accesses at runtime for the security-critical data. The properties inferred by a compiler-based static analysis of which instructions are allowed to write to a security-critical object according to the source code are checked and enforced at runtime. A pointer to an object can be manipulated by an attacker at runtime to reach another object by taking advantage of data layout arrangements in the object files, however, the compiler can infer from the source code that the pointer is not allowed to access the second object. In order to achieve high detection coverage, the proposed technique protects the entire dependence chain of instructions/data which contribute to the computation of the critical variable.

Our technique can also detect insider attacks in which parts of the program itself behave maliciously, or more specifically, against the source-code semantics of the original program. For example, assume that the browser contains a table of the user's public keys, which is marked critical. Now suppose the user loads a malicious plug-in which is allowed to execute code in the same memory space as the browser. Our technique will detect any attempt by the plug-in to write to the critical table independent of whether the plug-in code itself is checked since the security checks are in the browser.

We implemented and tested our technique on the OpenSSH server program. The results show that the overhead incurred by the runtime checking is highly dependent on the selected critical variable, ranging from 2.5% to 187%.

2. Related Work

Much of the earlier work related to memory corruption attacks has been targeted for specific attacks. For example, techniques such as StackGuard [4] and Libsafe [5] protect specifically

against buffer overflow vulnerabilities. Similarly, control flow checking techniques such as the No Execute Bit [6], Program Shepherding [1], and Control-flow Integrity [7] protect against attacks in which the attacker corrupts control data in the program. Recently, Chen et al. [2] have shown that it is possible for an attacker to overwrite non-control data (i.e., a password) in the system and exert the same level of influence over the application as if they had overwritten control data.

Another class of techniques, broadly called *information flow-based security*, enforces an externally imposed policy rather than one inherent to the program. Information flow-based security techniques [8] classify program data as high-security and low-security and ensure that low-security data cannot influence high-security data in the program. Sabel and Myers also explore information flow-based security, but focus on confidentiality rather than integrity/security [23]. A common variant of information security is taintedness (first proposed in the PERL programming language), which marks all externally supplied data (through user-input) as low-security data and ensures that these cannot influence high-security data in the program (such as pointers [9] and return addresses [10]). The main problem with these techniques is that they can result in false positives (rejection of valid code) and missed attacks due to incompatibilities or differences between the imposed policy and the policies inherent to the programming language.

Techniques such as *address space randomization* [11][12][13] are also based on the observation that an attacker exploits the disconnect between the source-level semantics and the application binary. However, they attempt to obfuscate the details of the underlying memory layout from the attacker rather than detecting and preventing attacks. Thus, protection is probabilistic and can be circumvented by repeated undetected attacks on the system, or through program information leaks such as pointer addresses exposed to the user or specially crafted format string attacks.

A broad class of techniques for ensuring *memory safety* of C and C++ programs has been proposed in the literature (e.g. [14][15]). These techniques use static analysis to prove pointers are safe at compile time and insert runtime checks for pointers that cannot be proven to be safe. These techniques are effective at guaranteeing protection from failures due to program errors. However, they cannot guarantee the program is secure from all malicious attackers since an attack may alter the data flow assumed by static analysis.

Another broad class of techniques generally referred to as *intrusion detection systems* has also been proposed [22]. The goal of these techniques is to detect security attacks by monitoring streams of network traffic. These techniques are useful for detecting well-known attacks for which attack invariants have been extracted and are advantageous because they incur little overhead since the network stream can be monitored by an independent computer. However,

there are two main disadvantages of these techniques. First, they can only detect attacks which are performed over the network and thus cannot detect physical/local attacks (such as smart-card attacks) or attacks in which a user modifies a local object, such as an environmental variable or program data file. Secondly, these techniques require certain known invariant byte strings (or at least some group of disjoint invariant bytes). Thus, intrusion detection systems can only defend against well-known attacks, or attacks which are very similar to previous attacks. Therefore, these techniques are not generally useful for against attacks which exploit a previously undisclosed vulnerability.

Recently a technique to guarantee memory safety in the presence of malicious attacks was proposed in [3]. The main idea is to compute the data flow graph of the program and enforce this data flow at runtime. This is performed by computing the reaching definitions of each variable in the program using intraprocedural, flow-sensitive analysis and ensuring that only the instructions that write to the memory location according to the reaching definitions analysis can do so at runtime. The attack model considered by this paper is similar to our paper in that the attacker is allowed to write to any memory location. It is assumed, however, that: (i) The attacker cannot overwrite variables stored in registers and hence checking memory loads and stores is sufficient to provide protection. (ii) The control flow of the program is preserved and can be enforced by instrumenting every read and write of program control data. Selectively protecting only the critical variables, as it is done in our approach, may violate and hence break the guarantees provided by the scheme. Further, our attack model assumes that the attacker can write to both memory and registers, and therefore, the checking is not restricted to memory loads and stores. Finally, our model does not require the original program's control flow to be preserved during an attack. The approach proposed in [3] does not consider context-sensitivity in computing the reaching definitions analysis which can introduce serious security holes in the data-flow graph computed statically.

3. Attack Model

We assume that the attacker can write to any location, be it memory or register at any point in the execution of the program. The attacker can also execute arbitrary code and change the control flow of the original program.

Recall that our goal is to protect certain data marked critical and prevent the attacker from corrupting this data against the semantics of the source program.

The attack model also covers physical attacks on the hardware e.g. smartcards, where the attacker can change the control flow of the program or change the operands fetched by an executing instruction at runtime. However, the attack model does not consider attacks in which

the executable image of the program is altered by the attacker, or attacks that tamper with the program executable before it is loaded. It is also assumed that the program load process is not in the control of the attacker and that a secure linker/loader is deployed.

In general, the technique proposed in this paper assures the integrity of critical data and not its confidentiality. Hence it does not address side-channel attacks [17]. However, unlike randomization based protection, the technique does not require the program to be free of information leaks (which would allow the attacker to guess the randomization details).

4. Concept of Information Flow Signature Checking

Our technique computes, for each critical data location in the program, the set of instructions that are allowed to write to the critical data location in the program (according to source code semantics). In this section, example attacks are used to illustrate how the signature checking technique detects attacks.

4.1. How Information Flow Signatures Detect Attacks

The program fragment given below prompts the user for a password and then compares the supplied password to the correct password stored in the `password` variable. If a user enters the correct seven-character password "asecret", the program outputs "Success". If the passwords do not match, the program outputs "Failed". Observe that the unchecked bounds on the `gets()` function allow the user to enter more than seven-characters, causing variables on the stack to be overwritten. Suppose that an attacker enters the string "attack! attack!". This would enable the attacker to overwrite the stack variable `password`.

```
1  int main()
2  {
3   char password[8] = "asecret";
4   char userpass[8];
5   printf("Enter Password:\n");
6   gets(userpass);
8   if(strncmp(userpass,password,7)==0)
9      printf("Success\n");
10   else
11     printf("Failed\n");
12 }
```

The main reason for this vulnerability is that the programmer never implied that the password buffer should be written to by the *gets()* function. From the source code, it is clear that the *gets()* function should only write to the *userpass* buffer; however, this is not enforced by the runtime system, which allows any instruction to write to any memory location. The attacker exploits this disconnect between the source-level semantics and the runtime layout of objects on the stack to

compromise the system. If the semantics of the source code were enforced, the *gets()* function would not be allowed to write to the password variable, and the attack would be prevented. Our method provides this enforcement for critical variables, such as the user password, which if corrupted can compromise the security of the application.

Detection. In the example, suppose that both the `password` and `userpass` buffers are critical. During static compilation, the compiler encodes the set of all instructions that are allowed to write to each of the critical variables. For the sake of clarity, signatures used in the examples are considered at the granularity of program statements rather than machine-level instructions. In this example, the only statement that is allowed to write to the `password` variable is `password[8] = "asecret"` on line 3, and the only statement that is allowed to write to the `userpass` variable is `gets(userpass)` on line 6. Suppose that the chosen signature is the set of all statements allowed to write to that variable. Thus, the signature for `password` is {3} and the signature for `userpass` is {6}. Now any attempt to write to `password` from within gets() on line 6 will be detected, since 6 is not in the signature for the `password` variable.

4.2. Why the signature must encode the entire dependence tree

In the example above, the signature included only those statements that directly manipulate the critical variable. In many programs, however, due to instruction/data dependencies, program variables can be altered indirectly following the instruction dependence chain. In order to provide protection against attacks which exploit an instruction/data dependence to tamper with the critical variables, it is necessary to encode the entire dependence tree. The example discussed in this section illustrates such a scenario.

```
1  int authenticate(char* username, char* password)
2  {
3      int authenticated=0;
4      int result;
5      char tmpbuf[512];
6      result = strncmp("asecret",password,7);
7       snprintf(tmpbuf,sizeof(tmpbuf),"user: %s",user);
8      tmpbuf[sizeof(tmpbuf)-1] = '\0';
9      syslog(LOG_NOTICE,tmpbuf);
10     authenticated=!result;
```

In the program above, assume that the variable `authenticated` is determined to be a critical variable. The signature of this statement is {10}, since the only instruction that writes to this statement is `authenticated=!result`. Suppose that the attacker attempts to overwrite the variable `authenticated` via manipulating statement 9 (for example, by exploiting a format string vulnerability). The signature technique, as described in Example 1 would detect tampering with `authenticated`. Now, assume that the attacker knows that `authenticated` is protected, and that only statement 10 can modify the variable without raising an alarm. Instead of directly overwriting

the authenticated variable, the attacker can overwrite the *result* variable, thus indirectly modifying *authenticated* and gaining access to the system. Notice that during the attack, the *authenticated* variable is only written by the statement on line 10, and thus the signatures match. Therefore, instead of simply encoding the direct dependencies of a critical variable in its signature, it is necessary to encode the entire dependence tree of the critical variable, thereby protecting critical variables from both direct and indirect modification.

5. Approach

This section describes the approach to derive information flow signatures for security-critical variables in applications. Compiler-based static program analysis is used to derive the signatures and hardware support is employed to enable runtime signature checking.

5.1. Static Analysis

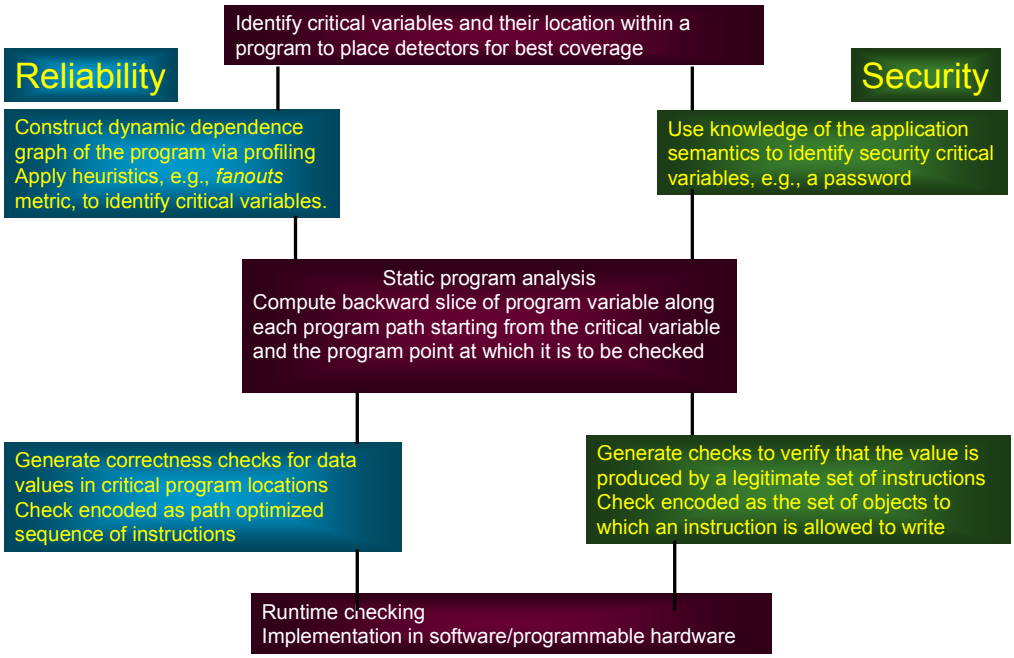

Figure 1: Steps in Compiler-based Static Analysis presented as part of a compiler-based security and reliability framework

5.1.1. Steps in Compiler Analysis

Figure 1 presents a unified compiler analysis framework to derive both error and attack detectors. This paper focuses on the derivation of attack detectors, which is shown on the right-hand side in Figure 1. These steps are as follows:

- The first step in compiler analysis is to determine the security-critical variables for the program. This can be done either by (i) the programmer, based on understanding of the program semantics (for example, variables used in authenticating a user in an SSH

server) or (ii) an automated tool, based on common targets for security attackers (for example, function pointers, return addresses, virtual function tables).

- The compiler then constructs the backward program slice for each of the critical variables starting from the program points where they are used in a security-critical context. This includes the uses of the critical variable in making control decisions and security-critical decisions (such as in a *strcmp()* function where a password is compared to the supplied system password). Ideally, this slice would be inter-procedural and take into account both register and memory dependences in the program.
- From the backward slice of a critical variable, instruction sequences can be extracted for each control-flow path in the slice. These instruction sequences are the dependence chains for the critical instruction and correspond to those instructions that can influence the value of a critical variable in a program execution.
- After computing the signatures, the compiler adds information to the program binary which is used to inform the runtime system of the critical variables and their pre-computed signatures. The compiler must also identify the places where the critical variables need to be checked (when they are used in a security critical fashion) and communicate this to the runtime system, using function calls or special instructions.

5.1.2. Data-Flow Analysis

In order to derive the signatures, the compiler needs to perform pointer and data-flow analysis on the program to determine the dependences for security critical variables. While data-flow analysis typically involves data dependences through registers (virtual or real), pointer analysis involves data-dependences through memory. Data-flow analysis attempts to answer the question of which instructions can directly or indirectly affect certain registers, while pointer analysis attempts to answer the question of which instructions can potentially write to specific memory variables (technically this is done by deriving *points-to* sets of pointer variables used by the write instructions). While data-flow analysis is a standard technique employed by most compilers (and is highly accurate), pointer analysis is a much harder problem and compilers typically perform various degrees of approximation depending on the space/time tradeoffs. The degree of approximation made by the compiler can affect the coverage of the derived signatures and allow attacks that could have been avoided in a more accurate analysis.

5.1.3. Pointer Analysis

This section provides a brief overview of the main factors that determine the precision of pointer analysis. A more detailed analysis is presented by [16].

- A context-sensitive analysis is one which isolates the calling contexts of called functions from each other. In a context-insensitive analysis on the other hand, values can flow from one call through the function and return to another caller. In other words, a context insensitive analysis does not distinguish one invocation of a function from another invocation.
- Flow-sensitive analysis considers the order of statements in a procedure when performing pointer analysis, and hence computes a solution for each program point. Flow-insensitive analysis, on the other hand, computes a solution assuming, conservatively, that any order of statements is possible in the procedure.
- The analysis feature known as field-sensitivity refers to how individual fields of a structure are modeled. Field-sensitive analysis considers each field of a struct or array separately, while field-insensitive analysis considers them to be a single object or location.
- The last type of pointer analysis considered is heap sensitivity. Heap insensitive analysis treats the entire heap as one single object. Heap sensitive analysis can minimally track heap objects by their allocation site, program location of calls to malloc(), in the program. While this suffices for many programs in which distinct objects are allocated at different sites, some programs may have customized allocation routines that allocate memory for multiple types of objects. In the latter case, an analysis that tracks heap objects by allocation site would in effect treat all objects allocated by the custom allocation routine as one object. Tracking the call sequences leading into the allocation site can further distinguish these objects and increase the level of heap sensitivity.

Each of these analyses increases the resolution and therefore sensitivity of our security checking scheme by decreasing the number of objects that alias each other. Utilizing a compiler that performs detailed pointer analysis significantly increases the amount of work an attacker needs to perform to gain access to the system.

5.2. Runtime tracking of signatures

The proposed signature checking technique enforces the properties of the compiler generated data dependence chain on critical variables at runtime. For efficiency and ease of implementation, the runtime enforcement of the signature checking scheme is divided into three levels. Together, these three levels provide the same guarantees provided by the signature scheme presented in the examples in section 2. Implementing the signature checking via this three-level scheme reduces the overhead and limits the propagation of compromised data. Before presenting the checking scheme, we introduce useful terminology:

Direct Critical Variable: A variable which has been determined either heuristically or by a human to be critical to the security of the application.

Trusted Instruction: An instruction which is statically determined to potentially read, write, or have influence on a critical variable directly or indirectly as defined by source code semantics. All the instructions in the dependence tree of the *direct critical variable* are marked trusted by the compiler.

Indirectly Critical Variable: A variable or object which influences the value of a *direct critical variable* (through a *trusted instruction*). All variables and objects in the data dependence tree of a *direct critical variable* are marked indirectly critical by the compiler.

Critical Bit: For each register and memory location, this bit is set if and only if the register or memory currently contains a *direct critical* or *indirectly critical variable*.

5.2.1. Three Level Checking Scheme

This section presents the three-level checking scheme and show the invariants and protection guaranteed by each level:

Level 1, the critical bit: The objective of the level 1 check is to separate the instructions and variables upon which any critical variable is dependent from the instructions and data which do not have influence on critical variables, according to compiler analysis. In the context of the dependence signature, level 1 ensures that instructions outside the signature of a critical variable do not influence instructions inside the signature. In order to accomplish this, a critical bit is maintained (by the hardware) for each register and memory location in the program. Initially, the critical bit for all variables that are used as operands to *trusted* instructions is set to 1. The propagation of the critical bit is performed by hardware in the background according to the rules presented in Table 1 and described below. An alarm is raised if a trusted instruction attempts to use a non-critical variable as an operand. This is because the compiler guarantees that all variables in the dependence tree of a direct critical variable are marked indirectly critical by initializing their critical bit to 1. Thus, under correct execution all operands of trusted instructions should be marked critical. If the critical bit of a trusted instruction's operand is not set, it means that the operand has been influenced, either directly or indirectly, by an instruction that the compiler determined should be unable to influence the operand. The fact that the operand has been influenced by an instruction outside of its dependence tree represents a violation of source code semantics, and therefore triggers an alarm. The runtime tracking of the critical bit is very similar to the Taintedness Tracking used in [18].

Propagation of the Critical Bit. The critical bit is propagated according to the simple rule shown below:

```
Crit(destination) ← Crit(operand 1) && Crit(operand 2) && Trusted(op)
```

The critical bit of the destination (register or memory) is set if and only if the critical bit of all operands is set and the instruction is trusted. If any one of these conditions does not hold, the critical bit of the destination is cleared.

It is important to point out that the actual checking of level 1 can be deferred and performed as part of the level 2 check. This is because an alarm is raised only in the event that a trusted instruction attempts to read from non-critical data. Since all trusted instructions are checked by level 2, it makes sense to do both checks at the same time, as shown in Figure 2. Thus, all that is required to enforce level one is the propagation of the critical bit according to straightforward rule presented above.

Table 1: Enumeration of all possible conditions and actions for the level 1 check

Instruction Type	Op	Data	Action	Reasoning
Trusted	Read	Critical	Does not trigger alarm	Trusted instruction is using valid data inside its dependence tree.
Trusted	Read	Non-Critical	Trigger Alarm	All dependent data of any trusted instruction is marked critical by the compiler, the cleared critical bit means an attacker has corrupted the data .
Trusted	Write	Critical	Allowed and passed to level 2	Trusted instruction is operating on valid data. A check needs to be performed to verify that the instruction writes to the dependence tree of the correct critical variable (level 2 check)
Trusted	Write	Non-Critical	Allowed and passed to level 2, set critical bit of target	Propagate the criticality to the destination if and only if all source operands for the trusted instruction are critical and the level 2 check passed.
Un-Trusted	Read	Critical	Allowed	Un-trusted instructions may be dependent on critical data, however critical data cannot be dependent on un-trusted instructions
Un-Trusted	Read	Non-Critical	Allowed	Not a security threat
Un-Trusted	Write	Critical	Reset critical bit	The critical bit of the destination must be cleared since this may be a malicious instruction.
Un-Trusted	Write	Non-Critical	Allowed	Not a security threat

Level 2, the sufficient condition. The level 2 check enforces two invariants of the dependence signature. First, it guarantees that instructions inside one signature chain cannot influence instructions or variables within another signature chain. Secondly, it guarantees that instructions inside a signature chain can only write to variables directly dependent upon them inside the signature chain.

Immediately performing the level 2 check instead of checking an accumulated dependence signature when the critical variable is reached allows us to detect attacks much earlier and

therefore prevent the propagation of corrupt data. Furthermore, performing an immediate check is more efficient than fetching an accumulated signature from memory, performing an update operation and then storing it back to memory.

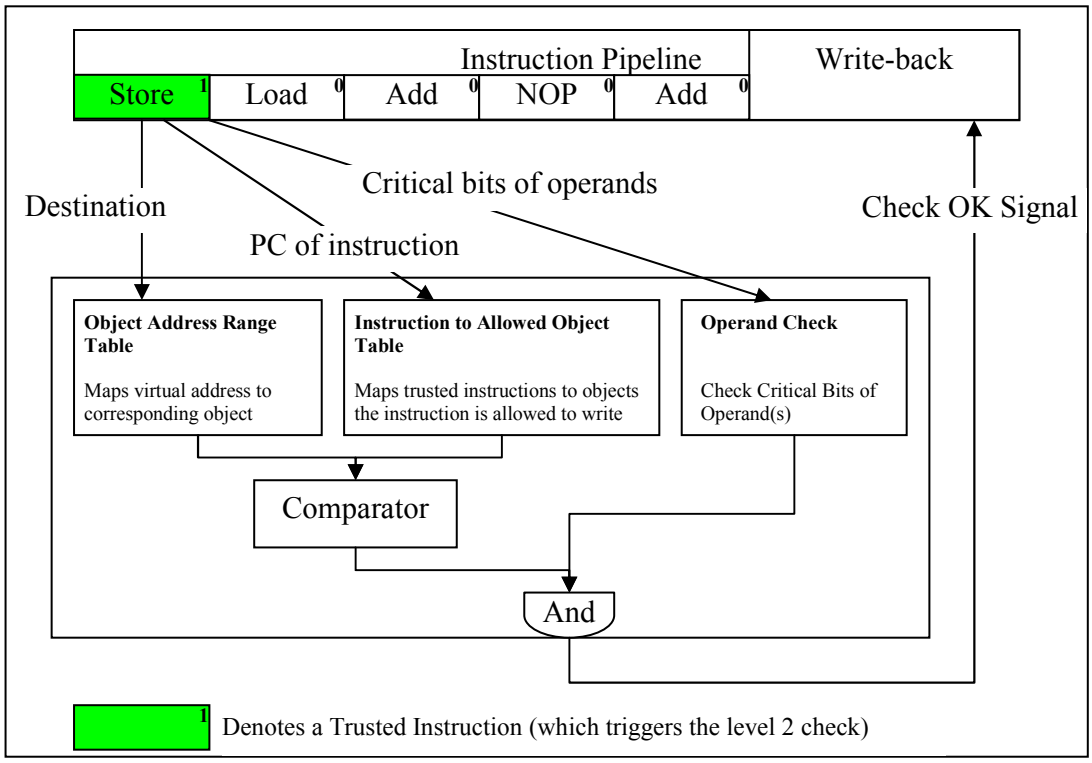

Figure 2: Theoretical Hardware Checking Implementation

As shown in Figure 2, the level 2 check is triggered by a trusted instruction. The hardware checker maintains two tables to facilitate the level 2 check:

- *Object Address Range Table* maps virtual addresses to the corresponding compiler generated object.
- *Instruction to Allowed Object Table* maps each trusted instruction to the objects that the compiler has determined the instruction is allowed to write to.

When the compiler encounters an instruction with a set trusted bit, the level 2 checking mechanism retrieves both the address of the instruction (stored in the program counter) and the destination address or register to which the instruction is attempting to write. The destination address is used to index the *Object Address Range Table* to determine which object the runtime instruction is actually attempting to modify. The program counter is used to index the *Instruction to Allowed Object Table* to determine which objects the current instruction is allowed to modify. The objects are then compared to ensure the instruction is writing to one of the objects in its allowed-write set. An additional check to determine whether the critical bit is set

for all operands (which was deferred from level 1) is performed in parallel. If either one of these checks fail, an alarm is raised before the instruction is allowed to enter the write-back stage in the processor.

Runtime Mapping of Objects. In order to perform the level 2 check, the hardware must have the ability to map a raw address to the corresponding symbolic object used by the compiler (accomplished using the object to address range table shown in Figure 2). The means by which this mapping is created depends on the type of object:

- *Global Variables, Static Variables or Constants*: These objects are statically allocated by the compiler at compile time and thus their mapping is known and can be provided to the hardware at initialization.
- *Local or Stack Variables:* The exact address of these variables is not known statically at compile time, but can be represented statically as an offset to the stack, or base pointer.
- *Dynamic or Heap Variables:* The address mappings of these variables must be determined and stored dynamically by intercepting calls to the heap allocator. This requires limited instrumentation of the memory related sys calls.

Level 3, the necessary condition. The goal of the level 3 check is to enforce that all instructions in a dependence chain of a critical variable are executed by the time that critical variable is used by a trusted instruction. This is necessary since in theory, an attacker could subvert the control flow of an application, somehow preventing the execution of a trusted instruction and thereby influencing the value of the critical variable. Another way of thinking about the level 3 check is that it ensures that every instruction in the signature chain of the critical variable is eventually checked.

The level three check can be implemented by statically extracting the set of all trusted instructions which have influence on the direct critical variable along each unique control flow path. These sets are then stored by the hardware. During runtime, the hardware keeps track of the set of all trusted instructions that have influenced each direct critical variable. Upon reaching a use of the direct critical variable, the runtime set for that direct critical variable is compared to the statically extracted sets. During correct execution, the runtime control flow path should correspond to one of the statically extracted control flow paths, resulting in a match. If an attacker manipulates the control flow to prevent the execution of certain trusted instructions, the sets will not match, and the attack will be detected. An example is provided in Figure 3:

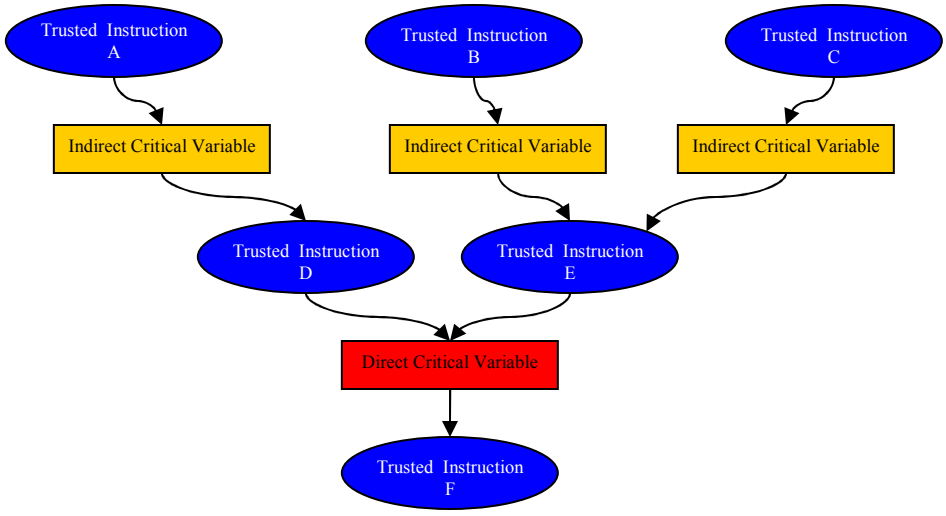

Figure 3: Example of the Level 3 Check

Suppose that in this program there are two control flow paths. Path 1 includes instructions A, D and F, whereas path 2 includes instruction B, C, E and F. During static analysis, the compiler will extract these two sets, {A, D, F} and {B, C, E, F}. Now, during runtime, suppose path 1 is executed. The runtime accumulator for the direct critical variable will add each instruction to the direct critical variable's set as it is executed. Thus, by the time instruction F is executed, the accumulator will contain {A, D, F} which will match the set extracted during static analysis. If the attacker prevents the execution of instruction D, the set will contain only {A, F} and thus will not match.

In order to implement this check, it is necessary to add an accumulator for each direct critical variable, as well as a means to track which trusted instructions have influence on which direct critical variables. The most straightforward way to implement the trusted instruction to direct critical variable mapping is to augment the *Instruction to Allowed Object Table*. Since each trusted instruction already has an entry in this table, all that is required is to add a field which contains the IDs of the direct critical variables that the instruction has influence on.

During the level 2 check, the hardware simply appends the current trusted instruction's id to the accumulators of each direct critical variable listed in the extra field of the *Instruction to Allowed Object Table.*

It is important to point out that the level 3 check implementation described above is most likely overkill. This is because it assumes the attacker can subvert the control flow of the application from any instruction, including in the middle of basic blocks. Without the ability to modify the instructions themselves, it is unclear how the attacker would subvert the control-flow in the middle of a basic block as this would require the creation of a control-flow instruction where non

exists. Thus, it should be sufficient to check for valid control flow at the basic block boundaries instead of within them. Quite a few schemes which do just that have been proposed, e.g. [7][19].

The three levels of checking are summarized by the diagram in Figure 4, which shows the actions performed by each check as well as the invariant provided by each level. Together the three levels of checking provide the same guarantees as the data-flow signature introduced in section 3 with the advantages of preventing the propagation of corrupted data and improved checking efficiency (in terms of performance).

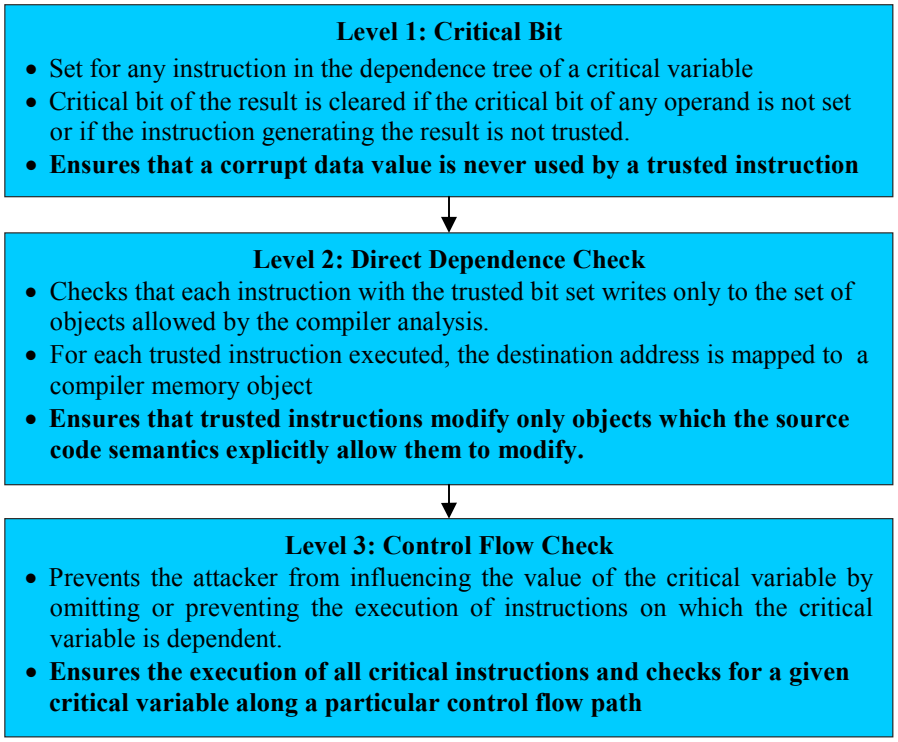

Figure 4: Steps in Compiler Analysis for Detector Derivation

Hardware Implementation. It is important to point out that in the case of a hardware implementation of the security checking scheme, no instrumentation of the program binary is required. Instead, at application load time, a relatively small configuration file can be associated with each executable protected by the technique. This configuration file would contain all the static variable mappings as well as the critical variables, indirectly critical variables, trusted instructions, and trusted instruction to allowed object mappings. This configuration file is then used to initialize the hardware checking engine. The use of a configuration file instead of direct executable instrumentation provides the unique ability to run the program at various security levels by utilizing multiple configuration files. For example, if an intrusion detection system detects an attack in progress, server daemons can be reloaded using configuration files which contain more critical variables, or perhaps critical variables specific to the attack at hand, thereby

providing greatly increased security. Thus, the technique allows the provided security to be configured based on application requirements.

6. Information Flow Signature Checking in Action

The authentication function for the OpenSSH server program is used to demonstrate the three-level checking scheme. This program is widely used to provide secure remote access to servers. Figure 5 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. Suppose an attacker can exploit a memory error in the *sys_auth_password* function to overwrite the *pw_passwd* variable with the encrypted version of their own password, allowing them to become authenticated by the system. The goal is to protect the encrypted password from the attacker by preventing the corruption of the *pw_passwd* variable. Therefore, the security critical variable is *pw_passwd* and the *strcmp* function which compares the encrypted password and the supplied password performs a security-critical use of the variable. The security checking technique guarantees that any corruption of the *pw_passwd* variable will be detected before it is used by the *strcmp* function.

```
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; /*Critical Variable Definition*/
        /* 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 the encrypted passwords match */
5:  return (strcmp(encrypted_password, pw_password) == 0);
}
```

Figure 5: Example code fragment showing derivation of attack detectors

Before discussing the technique, it is useful to consider the means by which an attacker can overwrite the encrypted password. One method would be for the attacker to corrupt *pw_passwd* directly through a memory error in *shadow_pw*, *strcmp*, or *xcrypt*. The other alternative is for the attacker to corrupt one of the values that *pw_passwd* depends on, and hope to influence the result of the final *strcmp* function. Our technique protects against both kinds of attacks

For simplicity, statements within the function body are annotated with integer labels that are used to derive the signature for the critical *pw_passwd* variable. These labels are also used in the

dependence graph presented in Figure 6. In reality, the signature is stored based on the addresses of the instructions corresponding to the statements. Each object or variable label maps onto multiple addresses and a runtime table precisely tracks which instruction address is allowed to modify which variables at runtime. The compiler performs inter-procedural slicing to find the origins of the variables being passed as function parameters and use this in the real signature (an example of the actual static dependence graph for the code in Figure 5 is presented in Figure 6).

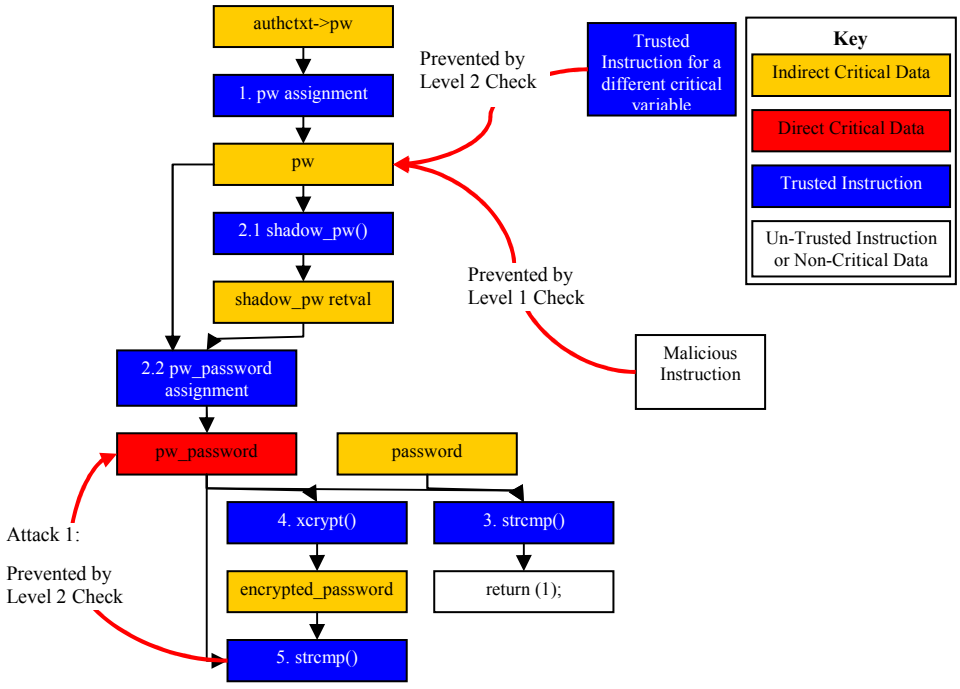

Figure 6: High-level static dependence graph corresponding to example code

Table 2: Mapping of Trusted Instructions to the data objects they are allowed modify

Trusted Instruction	Directly Modifiable Data Objects
1. *pw assignment	pw
2.1 shadow_pw()	shadow_pw retval
*2.2 *pw_password assignment*	pw_password
4. xcrypt()	encrypted_password
5. strcmp()	return value of sys_auth_passwd() function

7. Example Attack Scenarios

In this section, examples of real attack scenarios are used to illustrate the detection capabilities of Information Flow Signatures.

Attack 1: The attacker overwrites the *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 will be checked by the level 2 checking scheme. The checker will lookup instruction 5 in the trusted instruction table (presented in Table 2) revealing that the only object/variable that can be modified by instruction 5 is the return value of the *sys_auth_passwd()* function. Since the instruction has instead attempted to write to *pw_passwd*, the attack will be detected.

Attack 2: The attacker changes the *pw* pointer in *shadow_pw* in an attempt to influence the *pw_passwd* variable in Statement 3

Here, the attacker tries to overwrite the *pw* pointer instead of *pw_passwd* directly, possibly to cause *shadow_pw* to return a hash for which the password is known. In order to do this, the attacker must modify *pw* from a non-trusted instruction, since all trusted instruction are checked. However, since *pw* must be written to by a non-trusted instruction, its critical bit will be cleared. Now, when the trusted *strcmp* instruction is executed, it will attempt to use the *pw_password* variable, but since the critical bit is no longer set, an alarm will be raised and the attack will be detected.

Attack 3: The attacker corrupts the return value of *xcrypt*.

Context sensitivity allows us to detect attacks which cannot be detected by a context insensitive analysis such as that used by [3]. For example, an attack which could not be detected without context sensitivity is one in which the value returned by the *xcrypt* function is incorrect due to a memory error exploited by the attacker in the function. This is because a context insensitive analysis is less precise when tracking signatures inter-procedurally. With context sensitivity, the level 2 checks can be made more precise. For example, the checking engine can verify that the memory location pointed to by the return value of *xcrypt* is the one that should be written by that particular call of *xcrypt* and not a memory location that should be written during an unrelated invocation of *xcrypt*.

Attack 4: The attacker changes the 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. The attacker's goal in this attack is unclear, since he/she must still ensure that the *authctxt->pw* field holds the correct password in order to be authenticated by the system. Perhaps one could imagine a scenario in which *authctxt->valid* may be manipulated to the advantage of the attacker. If this attack were performed, it would be undetected by our scheme since modifying the *authctxt->valid* field makes the program execute a valid (but incorrect) control-path. This type of path dependence cannot be tracked by current compilers. It implies that each variable along the path is indirectly

dependent on any variable that was used in a branch decision leading to that path. If this were to be tracked, each variable would have a huge number of these dependencies since each possible branch decision would become a data dependency for each variable along that path.

8. Performance Evaluation

In order to evaluate performance of the proposed technique, the information flow signature checking scheme is applied to protect security critical variables in the authentication functions from the OpenSSH (secure shell) server program (introduced in Section 6). Four variables are identified (manually based on application semantics) as critical: (i) *authctxt,* a pointer to the user entered password (ii) *sys_auth_passwd*, the return value of the sys_auth_passwd() authentication function, the value of which determines whether the user is authenticated or not, (iii) *fakepw*, a pointer to a dummy SSH Authentication context used when the user has entered an invalid username, and (iv) *permit_empty_passwd*, a variable which determines if users are allowed to login through SSH if their password is empty. These variables are critical to the security of SSH because their corruption can result in the inaccurate authentication of an attacker. Information flow signature checking can detect tampering with the values of the selected variables and foil the potential attack.

Recall that the information flow signature encodes the entire dependence tree for a given critical variable. As a result, the performance overhead of the deployed technique depends on the length (in terms of instructions) of the dependence chain. The measurements reported in this section quantify this overhead in the context of the SSH application and the selected critical variables.

Application instrumentation. The *IMPACT* compiler [21] (developed at the University of Illinois) is employed for static analysis and *IMPACT's Lemulate* tool is used to simulate the hardware checks. Lemulate allows IMPACT to transform compiled output to a C-code representation rather than a program binary. This transformation allows each machine-level instruction to be represented using C syntax, with each line of the resulting C-code file corresponding to a single machine-level instruction. This C-code file can then be instrumented, re-compiled into a binary, and run. The hardware checks are simulated by creating a small C++ library to implement the two maps used by the level 2 check (*Object to Address Range Table* and *Instruction to Allowed Object Table*). The security checks are inserted by adding callbacks to the library before each trusted instruction in the C-code generated by Lemulate. The *check()* callback includes: (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.

Attacks are emulated by modifying the address to which an instruction writes along with the corresponding address in the *check()* callback. This represents the actual address (to which an instruction writes) being sent to the hardware checker at runtime.

Results.

Table 3 and Figure 7 report performance measurement results while targeting protection of selected critical variables. *Baseline* represents the time required (in microseconds) to run the compiled SSH authentication stub generated by Lemulate without any instrumentation. *Initialization* time represents the time required to add static variable mappings (for global variables and variables on the stack), in addition to time required to initialize the checker with the trusted instructions and their associated objects. *Checking + Dynamic* mapping time reports the time required to perform the runtime checks and the dynamic variable mappings (for objects allocated on the heap memory). The reported overhead is just an indirect measure of the overhead which would be incurred with a hardware implementation. With hardware support, the checks will be done in parallel with the main processor so few or no extra cycles will be incurred. Thus, although the results are useful to show the complexity of checking for various critical variables, they should not be interpreted as the overhead required for the checking scheme itself.

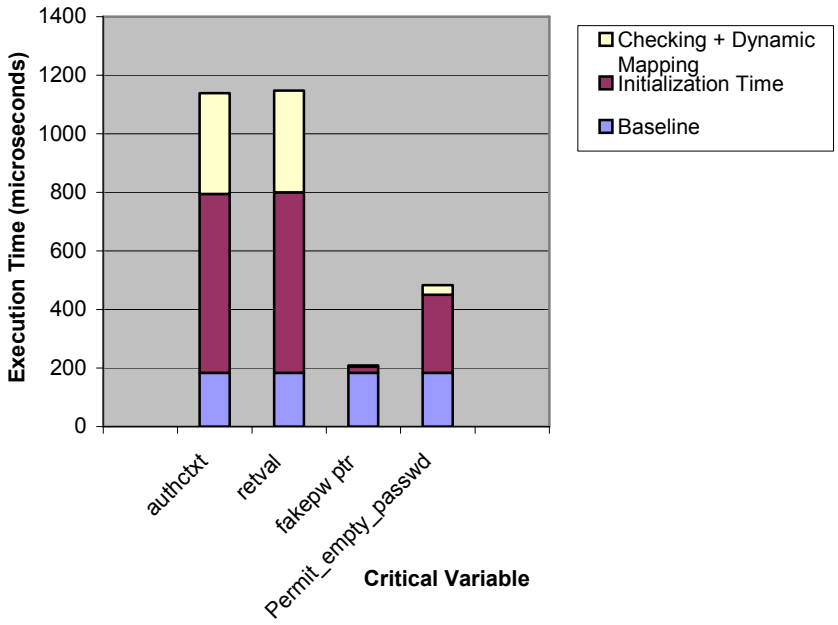

Figure 7: Checking Overhead Required to Protect Various Critical Variables

Table 3: Performance of Information Flow Signatures Deployed in OpenSSH

Critical Variable	Baseline	Initialization Time	Checking + Dynamic Mapping Time	Total (Full Instrumentation)
authctxt (pointer)	184	610	345	1139
sys_auth_passwd (return value)	184	616	348	1148
fakepw (pointer)	184	20	5	209
permit_empty_passwd (flag)	184	266	33	299

(Execution Times in usec)

Table 4: Proportion Instructions in Dependence Chains of Critical Variables of SSH Application

Critical Variable	Total Ops	Checked Ops	Local Variable Mappings	Global Variable Mappings
Authctxt	500	125	84	3
sys_auth_passwd	500	126	84	3
Fakepw	500	4	3	0
permit_empty_passwd	500	16	11	0

Protection of the *authctxt ptr* and the *sys_auth_passwd* critical variables incurs highest overhead of 2.9x ((184+348)/184 see Table 3). Note that the time required initialize the application is excluded since the overhead due to initialization is encountered only once when the application is first started. The significant overhead for these two variables is due to the fact that *authctxt* is the main authentication data structure and thus it is passed to each authentication function. Similarly, the *sys_auth_passwd* return value represents the final authentication decision, and ultimately determines whether the user is authenticated. As a result the dependence chains for these two variables are relatively long (see Table 4) and hence, the checking leads to higher overhead. Table 4 shows the numbers of instructions which need to be protected by information flow signatures for the four critical variables of the SSH application. One can see that for the *authctxt ptr* and the *sys_auth_passwd* critical variables, 25% and 25.2% of the total number of instructions in corresponding functions belong to the dependence chain of each variable, respectively. However, it is interesting to point out that these variables share the same 125 instructions and 84 indirect variables dependencies. This overlap between the dependence chains of these two critical variables is a significant advantage. For example, if the *authctxt* pointer variable is already being checked, the *sys_auth_passwd* variable can also be checked simply by adding a single trusted instruction.

Another interesting result is the significantly low overhead associated with checking the *fakepw* variable. This variable points to a dummy authentication context which SSH uses to authenticate against in the event that an invalid username is provided. However, if an attacker is able to overwrite the hash associated with this *dummy user* with a hash for which he/she knows the password, the attacker would be authenticated by certain authentication functions in the system (he/she would then have to overwrite the username to make it valid). Since very few legitimate instructions use this dummy authentication context, it is extremely efficient to check, and

arguably prevents an important security attack. In fact, this class of yet undiscovered but possible security vulnerabilities represents the strength of the information flow checking technique. Instead of attempting to find and patch the vulnerabilities, our technique actively protects critical data from a wide class of memory attacks, including undiscovered or unreleased exploits.

9. Conclusion

We have shown that the Dynamic Tracking of Information Flow Signatures is a powerful technique, providing security and attack detection for a very broad class of attacks. The technique is highly configurable, allowing the user to determine the desired level of protection, as well as which variables to protect. The technique uses detection of program data-flow violations as an indicator of malicious tampering with the system/application and prevents an attacker from exploiting the disconnect between source-level semantics and execution semantics of the program. Although the checking overhead incurred for certain critical variables is high, a true hardware implementation should reduce this overhead dramatically. A compile-time static program analysis is employed to extract a backward slice which collates all dependent instructions along each control-path used in computing the security critical program variables. Instructions (in terms of their PCs) identified in along the dependence chain are encoded to form a signature, which is checked at runtime. Any violation of the pre-computed signature raises an alarm. The approach is employed and demonstrated in the context of the SSH application. As part of future work, we plan to evaluate the technique using several other server-based applications such as WU-FTPD, httpd and sendmail. We also plan to create a hardware prototype in order to evaluate the overhead of a true hardware-based implementation.

10. References

[1] Kiriansky, V., Bruening, D., and Amarasinghe, S. Secure execution via program shepherding. In Proceedings 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. Securing Software by Enforcing Data-flow Integrity. Microsoft Research Cambridge. In OSDI, 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] Intel. Intel Itanium 2 Processor Reference Manual For Software Development and Optimization. Intel Corporation, 2004.

[7] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow Integrity: Principles, implementations, and applications. In ACM CCS (Nov. 2005).

[8] S. Z. Guyer, E. Berger, and C. Lin, Detecting errors with configurable whole-program dataflow analysis: Dept. Comput. Sci., Univ. Texas at Austin, Tech. Rep. TR 02-04, Feb. 2002.

[9] Chen, S.; Xu, J.; Nakka, N.; Kalbarczyk, Z.; Iyer, R.K. Defeating memory corruption attacks via pointer taintedness detection. In Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on 28 June-1 July 2005 Page(s): 378- 387

[10] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer. Architecture support for defending against buffer overflow attacks. EASY-2 Workshop, October 2002.

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

[12] 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.

[13] Berger, E. D. and Zorn, B. G. 2006. DieHard: probabilistic memory safety for unsafe languages. In Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (Ottawa, Ontario, Canada, June 11 - 14, 2006). PLDI '06. ACM Press, New York, NY, 158-168.

[14] 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.

[15] Dhurjati, D., Kowshik, S., and Adve, V. 2006. SAFECode: enforcing alias analysis for weakly typed languages. In Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (Ottawa, Ontario, Canada, June 11 - 14, 2006). PLDI '06. ACM Press, New York, NY, 144-157.

[16] 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, New York, NY, 113-123.

[17] Boneh, D., DeMillo, R. A., & Lipton, R. J. (2001). 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.

[18] 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.

[19] S. Bagchi et al. Hierarchical error detection in a software implemented fault tolerance (sift) environment. IEEE Transactions on Knowledge and Data Engineering, 12:203–224, March/April 2000.

[20] Dynamic Tracking of Information Flow Signatures for Security Checking. Tech Report. http://www.crhc.uiuc.edu/DEPEND/

[21] UIUC OpenIMPACT Effort. The OpenIMPACT IA-64 Compiler. http://gelato.uiuc.edu

[22] Newsome, J., Karp, B., and Song, D. 2005. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In Proceedings of the 2005 IEEE Symposium on Security and Privacy (May 08 - 11, 2005). SP. IEEE Computer Society, Washington, DC, 226-241.

[23] A. Sabelfeld, A. Myers. Language-based information-flow security. IEEE J-SAC, 2003