Securing software by enforcing data-flow integrity

Miguel Castro
Microsoft Research

Manuel Costa
Microsoft Research
University of Cambridge

Tim Harris
Microsoft Research

Abstract

Software attacks often subvert the intended data-flow in a vulnerable program. For example, attackers exploit buffer overflows and format string vulnerabilities to write data to unintended locations. We present a simple technique that prevents these attacks by enforcing data-flow integrity. It computes a data-flow graph using static analysis, and it instruments the program to ensure that the flow of data at runtime is allowed by the data-flow graph. We describe an efficient implementation of data-flow integrity enforcement that uses static analysis to reduce instrumentation overhead. This implementation can be used in practice to detect a broad class of attacks and errors because it can be applied automatically to C and C++ programs without modifications, it does not have false positives, and it has low overhead.

1 Introduction

Most software is written in unsafe languages like C and C++. Even programs written in type-safe languages have libraries and runtimes written in unsafe languages. Therefore, software is vulnerable to attacks and it is likely to remain vulnerable in the foreseeable future.

Almost all these attacks subvert the intended data-flow in the program. They exploit software vulnerabilities to write data to unintended locations. For example, *control-data attacks* exploit buffer overflows or other vulnerabilities to overwrite a return address [32], a function pointer [28], or some other piece of control-data. *Non-control-data attacks* exploit similar vulnerabilities to overwrite security critical data without subverting the intended control-flow in the program [14]. Non-control-data attacks have not been observed in the wild but they are just as serious and there are no good defenses against them. Non-control-data attacks will become common as we deploy defenses for control-data attacks.

This paper presents a technique that can prevent both control and non-control-data attacks by enforcing a simple safety property that we call *data-flow integrity*. This technique computes a data-flow graph for a vulnerable program using static analysis, and instruments the program to ensure that the flow of data at runtime is allowed by the data-flow graph. It can be applied to existing C and C++ programs automatically because it requires no modifications and it does not generate false positives.

There are many proposals to prevent attacks on software, for example, [30, 24, 27, 5, 18, 34, 31, 16, 19, 13, 37]. CCured [30] and Cyclone [24] propose memory-safe dialects of C that prevent all these attacks. The disadvantage of these approaches is that the effort to port existing C code to these dialects is non-trivial and they require significant changes to the C runtime, for example, they replace malloc and free by a garbage collector. There are several techniques that can be applied to existing programs but can only defend from attacks that overwrite specific targets, for example, return addresses [18], or that exploit specific types of vulnerabilities, for example, buffer overflows [25, 34]. Program shepherding [27] and control-flow integrity [5] provide a generic defense against control-data attacks but they cannot defend against non-control-data attacks. Techniques that perform dynamic taint analysis [37, 15, 19, 31, 13, 16, 22, 33] can prevent control-data attacks and they can prevent some non-control-data attacks [37, 16, 13], but they may have false positives and they incur a very high overhead without hardware support.

We implemented data-flow integrity enforcement using the Phoenix compiler infrastructure [29]. The implementation uses reaching definitions analysis [7] to compute a static data-flow graph. For each value read by an instruction, it computes the set of instructions that may write the value. The analysis relies on the same assumptions that existing compilers rely on to implement standard optimizations. These are precisely the assumptions that attacks violate and data-flow integrity enforcement detects when they are violated.

To enforce data-flow integrity at runtime, our implementation instruments the program to compute the definition that actually reaches each use at runtime. It maintains a table with the identifier of the last instruction to write to each memory position. The program is instrumented to update this table before every write and to prevent the attacker from tampering with the table. We also instrument reads to check if the identifier of the instruction that wrote the value being read is an element of the set computed by the static analysis. If it is not, we raise an exception. Our implementation does not generate false positives; when we raise an exception, the program has an error.

We have developed a number of optimizations to reduce the instrumentation overhead. The first optimization computes equivalence classes of instructions and assigns the same identifier to all the instructions in the same class. This reduces the number of bits required to represent identifiers and simplifies the code to check set membership on reads. Additionally, we perform static analysis of the low level intermediate representation of the compiler to remove unnecessary read and write instrumentation. This analysis is more conservative than the one used to compute the data-flow graph; it does not rely on any assumptions that may be invalidated by attacks.

We evaluated the efficacy and overhead of our implementation. The results show that data-flow integrity enforcement can prevent many control-data and non-control-data attacks, and that it can detect errors in existing programs. The instrumentation overhead is low: the space overhead is approximately 50%, and the average runtime overhead is between 44% and 103% in CPU intensive Spec 2000 benchmarks. In a Web server running Spec Web 1999, the runtime overhead is even lower: the average response time increases by 0.1% and peak throughput decreases by 23%. Thus, data-flow integrity enforcement can be used in practice to defend software from attacks.

2 Data flow integrity enforcement

This section starts by providing a high level overview of data-flow integrity enforcement. Then it describes in detail the static analysis and the instrumentation.

2.1 Overview

Data-flow integrity enforcement has three phases. The first phase uses static analysis to compute a data-flow graph for the vulnerable program. The second instruments the program to ensure that the data-flow at runtime is allowed by this graph. The last one runs the instrumented program and raises an exception if data-flow integrity is violated. We will use the simple example in Figure 1 to illustrate how these phases work.

```
1: int authenticated = 0;
2: char packet[1000];
3:
4: while (!authenticated) {
5:   PacketRead(packet);
6:
7:   if (Authenticate(packet))
8:     authenticated = 1;
9: }
10:
11: if (authenticated)
12:   ProcessPacket(packet);
```

Figure 1: Example vulnerable code in C.

Figure 1 shows a code fragment that is inspired by a vulnerability in SSH [3] that can be exploited to launch both control-data and non-control-data attacks [14]. In this example, we assume that `PacketRead` can write more than 1000 bytes to `packet` when it receives a packet from the network. This vulnerability could be exploited to overwrite the return address of the function or to overwrite `authenticated`. The first attack is a form of control-data attack that may allow the attacker to gain control over the execution. The second is a non-control-data attack that allows the attacker to bypass authentication and get its packet processed. Data-flow integrity enforcement can prevent both attacks.

We use reaching definitions analysis [7] to compute the static data-flow graph. Using the terminology from [7], an instruction that writes to a memory position *defines* the value in the memory position, and an instruction that reads the value is said to *use* the value. The analysis computes the set of reaching definitions for each use and assigns an identifier to each definition. It returns a map from instructions to definition identifiers and a set of reaching definition identifiers for each use, which we call the static data-flow graph.

For example, `authenticated` is used in lines 4 and 11. If we ran reaching definitions analysis in the source code, it might conclude that the definitions in lines 1 and 8 reach both uses. Therefore, the set of reaching definition identifiers for both uses would be $\{1, 8\}$, if we used the line numbers to identify the definitions.

The analysis can be imprecise but it is important that it be conservative. It must include in the set all definitions that may reach a use at runtime but it may include additional definitions. For example, only the definition in line 8 can reach the use in line 11 but the analysis might compute the set of reaching definitions $\{1, 8\}$. This ensures that imprecisions can lead to false negatives but not false positives: data-flow integrity enforcement may miss

some attacks but it will never signal an error unless there is one. We believe that this property is very important because users are not likely to deploy security solutions that can break running systems in the absence of errors.

The second phase instruments the program to enforce a simple safety property that we call *data-flow integrity*, i.e., *whenever a value is read, the definition identifier of the instruction that wrote the value is in the set of reaching definitions for the read (if there is one).*

The program is instrumented to compute the definition that reaches each read at runtime and to check if this definition is in the set of reaching definition identifiers that was computed statically. To compute reaching definitions at runtime, we maintain the *runtime definitions table* (RDT) that records the identifier of the last instruction to write to each memory position. Every write is instrumented to update the RDT. The instrumentation before reads uses the address of the value being read to retrieve the identifier from the RDT. Then, it checks if this identifier is in the statically-computed set. For example in Figure 1, we would add code to set RDT[`&authenticated`] to 8 in line 8, and to check if RDT[`&authenticated`] $\in \{1, 8\}$ in lines 4 and 11.

We want to enforce data-flow integrity in the presence of a strong attacker that can write anywhere and that can even execute data. To achieve this goal, the attacker must be prevented from tampering with the RDT, tampering with the code, or bypassing the instrumentation.

RDT tampering is prevented by instrumenting writes to check if the target address is within the memory region allocated to the RDT. Any attempt to write to the RDT generates an exception. Code tampering can be prevented with the same checks or by using read-only protection for code pages; we use the latter, which is already available in most processors.

To prevent the attacker from bypassing the instrumentation, we must prevent tampering with the target addresses of indirect control transfers. We instrument writes and reads of programmer-defined control-data as described before. In addition, we instrument writes and reads of control-data added by the compiler in the same way. For example, we set the RDT entry for a return address to a well-known value and we check if the entry still holds the value on return. In our test cases, this instrumentation is sufficient to prevent the attacker from tampering with the target addresses of indirect control transfers. However, there may be cases where the reaching definitions analysis is not precise enough. For example, the reaching definitions set for the read of a function pointer may contain the identifier of an array store instruction that an attacker can use to overwrite the function pointer. We can detect when this happens and use existing techniques to ensure control-flow integrity [27, 5].

It is possible to trade off coverage for lower overhead by removing reaching definitions sets for some uses: some attacks may go undetected because those uses are not instrumented but the overhead will be lower. We can still ensure data-flow integrity for all other uses provided uses of control-data are instrumented. We experimented with a variant of our implementation that only instruments uses of control-data and uses of local variables without definitions external to the function. This variant is interesting because it has low overhead and it can still catch many interesting attacks. For example, it can prevent any attack that violates control-flow integrity and the attack in our example.

When the instrumented program runs, any deviation from the data-flow graph computed statically raises an exception. Since the analysis is conservative, there are no false positives. If there is an exception, the program has an error. The error may or not be triggered by an attack. In addition to detecting various known attacks, we found several unknown errors in our test cases.

2.2 Static analysis

We compute reaching definitions using a combination of two analyses: a flow-sensitive intra-procedural analysis and a flow-insensitive and context-insensitive inter-procedural analysis. We implemented both analyses using the Phoenix compiler framework [29]. They operate on Phoenix's high level intermediate representation (HIR), which enables them to be applied to different languages and different target architectures. Figure 2 shows the HIR for the vulnerable C code in Figure 1.

The intra-procedural analysis takes flow control into account. It is implemented by traversing the static single-assignment representation [10] produced by Phoenix. We use this analysis to compute reaching definitions for uses of local variables that have no definitions external to the function in which they are declared. The inter-

```
      _authenticated = ASSIGN 0                #1
$L6:  t274   = CMP(NE) _authenticated, 0       #2
              CBRANCH(NE) t274, $L7, $L8       #3
$L8:  tv275 = CALL &_PacketRead, &_packet      #4
      t276  = CALL &_Authenticate, &_packet    #5
      t277  = CMP(NE) t276, 0                  #6
              CBRANCH(NE) t277, $L10, $L9      #7
$L10: _authenticated = ASSIGN 1                #8
$L9:          GOTO $L6                         #9
$L7:  t278   = CMP(NE) _authenticated, 0       #10
              CBRANCH(NE) t278, $L12, $L11     #11
$L12: tv279 = CALL &_ProcessPacket, &_packet   #12
$L11:
```

Figure 2: Example vulnerable code in high-level intermediate representation (HIR).

procedural analysis is used to compute reaching definitions for all other uses.

The inter-procedural analysis is less precise to allow it to scale to large programs. It ignores control-flow and it does not take the calling context into account when analyzing functions. We implemented Andersen's points-to analysis [9] to compute the set of objects that each pointer can point to, and we use these points-to sets to compute reaching definitions. The implementation is similar to the one described in [21] but it is field-insensitive rather than field-based (i.e., it does not distinguish between the different fields in a structure, union, or class). These imprecisions can lead to false negatives, for example, we may fail to detect an attack that overflows a buffer in a structure to overwrite a security critical field in the same structure. We plan to implement an analysis with better precision in the future (e.g., [12]).

The points-to analysis makes a global pass over all source files to collect *subset constraints*. Each assignment $x = y$ results in a subset constraint $x \supseteq y$, which means that the set of possible values of x contains the set of possible values of y. The analysis uses Phoenix to compile each source file to HIR and it writes all subset constraints in the HIR to a file. After this global pass, it computes the points-to sets by iterating over all the constraints until it reaches a fixed point. Then, it stores the points-to sets in a file.

During the global pass, we also collect the target locations and identifiers of instructions that write to locations that may be read in other functions. These include writes to locations obtained by dereferencing pointers, to static and global variables, and to local variables whose address is taken. This information is also written to a file.

We compute inter-procedural reaching definitions using the points-to sets and information about write instructions collected during the global pass. For uses of variables and temporaries, the set of reaching definitions is the union of the set containing the identifiers of all writes to the variable (or temporary) with the sets containing the identifiers of all writes to dereferences of pointers that may point to the variable (or temporary). For pointer dereferences, the set of reaching definitions is the union of the set containing the identifiers of all writes to the dereferenced pointer with the sets of reaching definitions of all the variables the pointer can point to. The sets of inter-procedural reaching definitions are written to a file that is used to instrument the program.

Both the intra-procedural and the inter-procedural analyses assume that the relative layout of independent objects in memory is undefined [12]. They assume that correct programs do not use pointer arithmetic to navigate between independent objects in memory. For example in Figure 2, the analyses assume that correct programs will not use a pointer to the `packet` array to write to the `authenticated` variable. Existing compilers already make this assumption when implementing several standard optimizations. Therefore, this assumption applies to the vast majority of programs. However, it is precisely this assumption that is violated by most attacks. Data-flow integrity enforcement detects and prevents these attacks.

We applied the analysis to the sample code in Figure 2 using the numbers at the end of the lines to identify definitions. The set of reaching definitions is $\{1, 8\}$ for both uses of `authenticated` (in lines 2 and 10). The temporaries have a single definition in the preceding instruction. Since we control code generation, we can ensure that these temporaries are placed in registers beyond the reach of an attacker. The attacker cannot violate data-flow integrity by overwriting these registers because our instrumentation prevents it from subverting the control flow. We only compute reaching definitions and instrument accesses of temporaries that are spilled to memory.

2.3 Instrumentation

We add instrumentation by inserting new high-level instructions into the HIR of the program. The instructions have the form:

```
SETDEF opnd id
CHECKDEF opnd setName.
```

The first instruction sets the RDT entry for `opnd` to `id`. The second retrieves the runtime definition identifier for `opnd` from the RDT and checks if the identifier is in the reaching definitions set with name `setName`. The compiler maintains a map from set names to set values that is used when lowering `CHECKDEF` instructions to the assembly of the target machine. Instrumenting a high-level representation of the code has the advantage of making the instrumentation machinery independent of the source language and mostly independent of the target architecture. Currently, we only target the x86 architecture but it would be simple to target other architectures.

Figure 3 shows the HIR for the vulnerable code with high-level instrumentation generated from the information computed by the reaching definitions analysis. The set with name 100 has the value $\{1, 8\}$. We do not instrument temporaries that we can ensure are allocated to registers, and we also do not instrument the uses of `&packet` because addresses of local variables are computed by adding a constant to the frame pointer.

Before describing how the high-level instrumentation is lowered to assembly, we need to describe how we implement the RDT. To enable efficient accesses, the RDT is implemented as an array with a definition identifier for each 32-bit memory word in the instrumented program. Each definition identifier is two bytes long, which seems sufficient even for large programs. This results in a space

```
      SETDEF _authenticated 1
      _authenticated = ASSIGN 0              #1
$L6:  CHECKDEF _authenticated 100
      t274  = CMP(NE) _authenticated, 0      #2
              CBRANCH(NE) t274, $L7, $L8     #3
$L8:  tv275 = CALL &_PacketRead, &_packet    #4
      t276  = CALL &_Authenticate, &_packet  #5
      t277  = CMP(NE) t276, 0                #6
              CBRANCH(NE) t277, $L10, $L9    #7
$L10: SETDEF _authenticated 8
      _authenticated = ASSIGN 1              #8
$L9:          GOTO $L6                       #9
$L7:  CHECKDEF _authenticated 100
      t278  = CMP(NE) _authenticated, 0      #10
              CBRANCH(NE) t278, $L12, $L11   #11
$L12: tv279 = CALL &_ProcessPacket, &_packet #12
$L11:
```

Figure 3: Example vulnerable code in HIR with instrumentation.

overhead of approximately 50%.

There are some subtle issues on the choice of memory granularity for recording definition identifiers. Since programs can access memory at byte granularity, it would seem necessary to record a 2-byte definition identifier for every byte of memory. This would result in a space overhead of approximately 200%, which may not be practical. We are able to record a single identifier for each 32-bit word because we can generate code in which no two variables with distinct reaching definition sets share the same aligned 32-bit memory word. Since our reaching definitions analysis does not distinguish between different fields in objects and between different elements in arrays, it is not necessary to change the layout of arrays and objects. We only changed the compiler to use a minimum alignment of 32 bits when laying out local variables in a stack frame and globals in the data segment. Function arguments and heap allocated objects were already appropriately aligned.

In the current implementation, we allocate the lowest 1GB of the virtual address space to the program being instrumented and 512MB to the RDT with a guard page between them, that is, the guard page is at address 40000000h and the base of the RDT is at address 40001000h. So, to compute the address of the RDT entry for an operand, we simply take the address of the operand shift it right by two, multiply the result by two, and add 40001000h. This layout also enables efficient bounds checking of the target addresses of writes to prevent tampering with the RDT: we raise an exception if the bitwise and of the target address with c0000000h is non-zero (as in [39]). The guard page allows us to check only the target address for the write and ignore the size.

The high-level instrumentation is lowered to x86 assembly as illustrated by the following examples. We lower `SETDEF authenticated 1` to:

```
    lea  ecx,[_authenticated]
    test ecx,0C0000000h
    je   L
    int  3
L:  shr  ecx,2
    mov  word ptr [ecx*2+40001000h],1
```

The first instruction loads the target address of the write into `ecx` and the following three instructions perform the bounds check on the address. If the check fails, we currently generate a breakpoint (`int 3`), which is very convenient for debugging. Another exception would be more appropriate in production use. The `shr` instruction is used to compute the address of the RDT entry for `_authenticated` and the `mov` instruction updates the entry. If the size of the operand is greater than 32 bits, it is necessary to update the entries in the RDT corresponding to the other words. We can update entries for 64-bit operands with a single `mov` instruction by moving the concatenation of two copies of the identifier. But we add additional instructions with larger operands.

The `CHECKDEF authenticated 100` instruction is lowered to:

```
    lea  ecx,[_authenticated]
    shr  ecx,2
    mov  cx, word ptr [ecx*2+40001000h]
    cmp  cx,1
    je   L
    cmp  cx,8
    je   L
    int  3
L:
```

This code compares the definition identifier in the RDT entry for `_authenticated` with the definition identifiers in set 100. When the operand is larger than 32 bits, we add additional comparisons for the other words.

In addition, we instrument definitions and uses of control-data introduced in the compilation process. For example on function entry, we add the following code to set the RDT entry corresponding to the function's return address to zero:

```
    mov ecx, esp
    shr ecx,2
    mov word ptr [ecx*2+40001000h],0
```

Just before returning, we add code to check if the definition identifier that reaches the use of the return address is zero.

```
    mov ecx, esp
    shr ecx,2
    cmp word ptr [ecx*2+40001000h],0
```

```
    je   L
    int  3
L:  ret
```

Going back to our example, the instrumented code is no longer vulnerable to either the control-data attack that overwrites the return address or the non-control-data attack that overwrites `authenticated`. Since the analysis concludes that `authenticated` is not aliased with `packet`, writes to `packet` have identifiers that are guaranteed to be distinct from 1 or 8. Additionally, the identifier zero is only used on function entry for return addresses. Therefore, any return address overwrite would also be detected.

2.4 Runtime environment

Programs rely on a complex runtime environment that includes several libraries. It is often impossible to analyze the source code of these libraries. Frequently, only the binaries are available and, even when source code is available, some functions are hand-written in assembly. Yet, many attacks make use of libraries when exploiting vulnerabilities. For example, string manipulation functions in the C library are notorious for their use in exploits of buffer overflow vulnerabilities.

Techniques that use source code analysis to instrument writes fail to provide any guarantees unless library calls are wrapped to perform safety checks. These techniques include array bounds checking [34, 25] and memory-safe dialects of C [30, 24]. Sometimes wrappers are also required to perform memory layout conversions [30]. Writing these wrappers can be onerous.

Data-flow integrity enforcement does not require any wrappers for library functions that are not analyzed. If a programs calls these functions, we cannot instrument some uses but we guarantee integrity of the data-flow for all other uses. To do this, we instrument library binaries to set the RDT entries for any memory they write to an invalid definition identifier (which can be done without the source code). To prevent false positives, we do not instrument uses of memory that is reachable from a pointer that is passed to or received from one of these library functions. We determine these uses by running a simple reachability analysis on the output of our points-to analysis. We instrument all other uses. In particular, we can always instrument all uses of local variables without definitions external to the function where they are declared, which prevents the attack in our example.

We provide the option to define library wrappers to increase coverage. To define a wrapper for a library function, one must write a wrapper function and describe the subset constraints that calling the function adds to the points-to analysis. The wrapper function checks definition identifiers for the memory read by the library, calls the library, and sets definition identifiers for the memory written by the library. We instrument the code to call the wrapper instead of the original function and to supply the wrapper with reaching definition sets for the memory read by the library function and a definition identifier for the memory it writes. For example, a wrapper for the Windows NT operating system call `CreateProcess` can check the integrity of the application name and command line strings supplied as arguments.

```
void*
Dfi_memcpy(int** defArgs, void *dest,
           const void *src, size_t count)
{
  unsigned int defId = (unsigned) defArgs[0];
  CHECK_BOUNDS(dest, count);
  CHECK_ARRAY(defArgs,1,src,count);

  memcpy(dest, src, count);

  UPDATE_RDT(dest, count, defId);

  return dest;
}
```

Figure 4: Example library function wrapper.

We have written wrappers for library functions used in our test cases, which include some operating system calls. Figure 4 shows an example wrapper for `memcpy`. `CHECK_BOUNDS` ensures that `memcpy` does not write into the RDT and `CHECK_ARRAY` checks if the identifiers in the RDT for the bytes in `src` are in the reaching definitions set supplied in `defArgs`. The RDT entries for the bytes written to `dest` are set to `defId` by `UPDATE_RDT`.

3 Optimizations

A naïve implementation of data-flow integrity enforcement can perform poorly: each definition introduces a write to the RDT and each use check introduces a read from the RDT followed by comparisons against each identifier in the set of reaching definitions for the use. This section discusses a number of techniques that we have developed to reduce this overhead.

3.1 Renaming equivalent definitions

The first optimization partitions definitions into equivalence classes in a way that allows us to safely assign the same identifier to all definitions in the same class. Two definitions are *equivalent* if they have exactly the same set of uses. This reduces the number of comparisons in `CHECKDEF`s and the number of bits required to represent identifiers. For example, both definitions of `_authenticated` in Figure 2 have the same set of uses

computed by the static analysis. We assign the same identifier 1 to both definitions. Therefore, `CHECKDEF authenticated 100` requires only one comparison. It is compiled to:

```
    lea  ecx,[_authenticated]
    shr  ecx,2
    cmp  word ptr [ecx*2+40001000h],1
    je   L
    int  3
L:
```

Our experiments show that this optimization is fundamental to obtain low overhead.

3.2 Removing bounds checks on writes

We check the target addresses of writes to prevent the attacker from tampering with the RDT. We can optimize `SETDEF`s by removing these checks from all *safe* writes. In the current implementation, a write is safe if the target address is obtained by adding a small constant offset (possibly zero) to the stack pointer, frame pointer, or to the address of a global or static variable. We require the sum of the offset and the size of the data being written to be less than 4KB (which is the size of the guard page that we allocate before the RDT).

For example in Figure 3, since `_authenticated` is a local variable whose address is obtained by adding a small constant to the frame pointer, we can remove the bounds check from `SETDEF authenticated 1`. The `SETDEF` is compiled to:

```
    lea  ecx,[_authenticated]
    shr  ecx,2
    mov  word ptr [ecx*2+40001000h],1
```

3.3 Removing `SETDEF`s and `CHECKDEF`s

The next optimization runs data-flow analysis to remove some `SETDEF`s and `CHECKDEF`s. We must be careful to avoid two problems. First, we must not rely on high level analyses whose inferences are unsound once data-flow integrity has been lost: the entire purpose of the instrumentation is to detect cases where the program's data-flow integrity is compromised. Second, we must not remove checks too early during compilation because later code transformations may change the situations in which data-flow integrity is lost. Therefore, we perform our optimizations when `SETDEF` and `CHECKDEF` operations are still present in their HIR form but the remainder of the program has already been lowered to the native instruction set and is ready to emit.

For simplicity, we will describe the case of instruction sequences within the same basic block. Our implementation deals with the case of propagating flow variables into a basic block if it has exactly one predecessor. It could be extended to provide more optimization opportunities by using the techniques in [8, 20].

Our instrumentation is redundant in the following cases. Suppose that instructions I_1 and I_2 are a pair of `SETDEF`s or `CHECKDEF`s *relating to the same data* that execute without any intervening write to that data.

1. If I_1 and I_2 are both `SETDEF`s with the same identifier then I_2 is redundant.
2. If I_1 and I_2 are both `SETDEF`s with no intervening `CHECKDEF` for that data then I_1 is redundant.
3. If I_1 is a `SETDEF` for ID_1 and I_2 a `CHECKDEF` for a set containing ID_1 then I_2 is redundant (indeed, ID_1 must be in I_2's set if the data-flow analysis was performed correctly).
4. If I_1 and I_2 are both `CHECKDEF`s against sets IDS_1 and IDS_2 respectively then IDS_2 can be reduced to contain only elements in IDS_1 (the earlier check guarantees no other elements are present). Furthermore, if IDS_1 and IDS_2 hold identical elements then I_2 can be removed (it is not possible for the latter check to fail if the earlier check succeeds).
5. If I_1 is a `CHECKDEF` against set IDS_1 and I_2 a `SETDEF` for ID_2 then I_2 is redundant if $IDS_1 = \{ID_2\}$.

In practice rules 3 and 4 are the most effective. Rule 3 eliminates many `CHECKDEF` instructions when uses of data occur close to their definitions. Rule 4 lets us remove `CHECKDEF` instructions where the same data is used repeatedly, even if there are aliasing writes between the data's definition and the first of those uses.

To identify redundant instrumentation, we use symbolic execution of the native code augmented with `SETDEF` and `CHECKDEF` operations. We update the symbolic state of the registers after each instruction and the symbolic state of the RDT after `SETDEF`s and `CHECKDEF`s. The symbolic state of the RDT maps symbolic memory addresses to sets of definition identifiers.

The current implementation uses a simple test to compare symbolic addresses. Two addresses are equal if they are syntactically equal. They are different if they are computed by adding different offsets to the same symbolic register state. Otherwise, they may refer to aliased memory locations. A write to memory invalidates the symbolic state of a register if this state refers to the contents of a memory position that may be aliased with the write's target. Additionally, it removes mappings for any memory that may be aliased with the write's target from the symbolic RDT state. We apply the rules to eliminate redundant instrumentation after each `SETDEF` and `CHECKDEF` by examining the symbolic RDT state.

3.4 Optimizing membership checks

Another optimization renames definitions to reduce the cost of membership checks in CHECKDEFs. Membership checks can be implemented more efficiently when sets contain ranges of consecutive identifiers: a check against $\{0 \ldots n\}$ can be implemented by a single unsigned integer comparison against n, and a check against $\{n \ldots m\}$ can be implemented by subtracting n and performing an unsigned comparison against $m - n$.

We define the *cost* of a CHECKDEF as the number of subtractions and comparisons necessary to perform its membership check. For instance, the cost of checking membership in $\{1, 3, 7\}$ is three, but it is only one for $\{0, 1, 2\}$. We say that the *total cost* of an identifier set is the cost of a single CHECKDEF against it, multiplied by the number of CHECKDEFs against it that occur in the program text.

We use a simple greedy algorithm to attempt to reduce the sum of the total costs of all sets: we sort the sets in order of decreasing total cost and proceed to assign the most costly sets to contiguous identifier ranges. We start from identifier 0 and so the set with the largest total cost benefits from the cheapest comparisons. Once we have started assigning identifiers to the elements of popular sets, we are constrained in the assignment of identifiers to elements in less popular sets that intersect them.

3.5 Removing SETDEFs for safe definitions

The last optimization identifies local variables that have no definitions outside the function and that are written only by safe writes (according to the definition of safe in 3.2). It replaces all SETDEFs for such a variable by a single SETDEF with identifier 0 that is placed on function entry. It also simplifies CHECKDEFs for such variables to have a reaching definitions set of $\{0\}$. This is sufficient because it detects violations of data-flow integrity that overwrite these variables and safe writes cannot overwrite other variables.

3.6 Other optimizations

We also tried to reorder comparisons in CHECKDEFs to take advantage of the fact that different definitions can reach a use with different frequencies at runtime. However, this optimization had little impact in our sample programs after we applied the other optimizations. Therefore, it is not included in the results that we present.

We have focused on optimizations that can be made without reference to profiling data. Using feedback, whether dynamically or from full program runs, is an interesting direction for future work.

4 Evaluation

We ran experiments to evaluate the overhead of our implementation and the effectiveness of data-flow integrity enforcement at preventing control-data and non-control-data attacks. This section presents our results.

4.1 Overhead

We used several programs from the SPEC CPU 2000 benchmark suite to measure the overhead added by our instrumentation. We chose these programs to facilitate comparison with other techniques that have been evaluated using the same benchmark suite, for example, [5]. We ran six integer benchmarks (gzip, vpr, mcf, crafty, bzip2, and twolf) and three floating point benchmarks (art, equake, and ammp).

These benchmarks are CPU-intensive and they spend most time executing instrumented code at user level. The overhead of our instrumentation is likely to be higher in these benchmarks than in other programs where it would be masked by other overheads. Therefore, we also measured the overhead in a Web server running the SPEC WEB 1999 benchmark (see Section 4.1.4).

We compared the running time and peak physical memory usage of the programs compiled using Phoenix with and without our instrumentation. The version without instrumentation is compiled with optimization (/Ox) but the current release of Phoenix does not implement many optimizations that are available in other compilers.

We ran the experiments on an idle Dell Precision Workstation 350 with a 3GHz Intel Pentium 4 processor and 2GB of memory. The operating system was Windows XP professional with service pack 2. We ran each experiment three times and we present the average result. The variance in the results was negligible.

4.1.1 Comparison with uninstrumented code

The first set of experiments measured the overhead of data-flow integrity enforcement with all the optimizations. We evaluated two variants of data-flow integrity enforcement: *intraproc DFI* only instruments uses of control-data and uses of local variables without definitions outside their function, and *interproc DFI* is the variant described earlier in the paper.

Figure 5 shows the execution time of the two variants of data-flow integrity enforcement normalized by the execution time without instrumentation (which is labeled *base* in the figure). The average overhead is 43% for *intraproc DFI* and 104% for *interproc DFI*. We believe that the overhead of either variant is sufficiently low to enable the use of data-flow integrity enforcement in applications that are not CPU intensive or that spend more

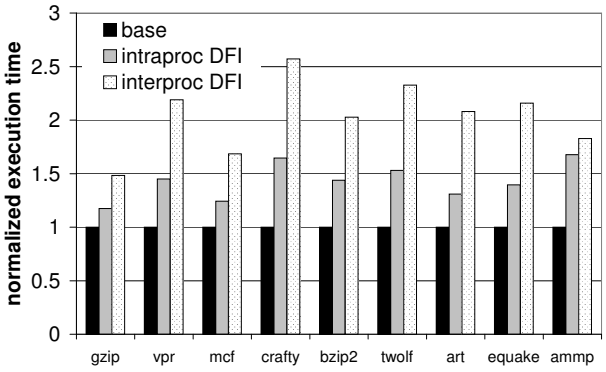

Figure 5: Execution time with data-flow integrity enforcement (normalized by the execution time without instrumentation).

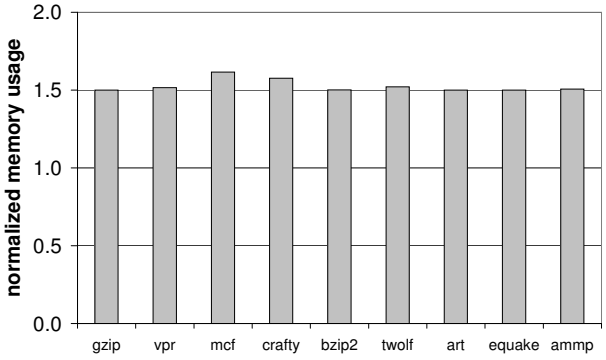

Figure 6: Space overhead of data-flow integrity enforcement relative to the execution without instrumentation.

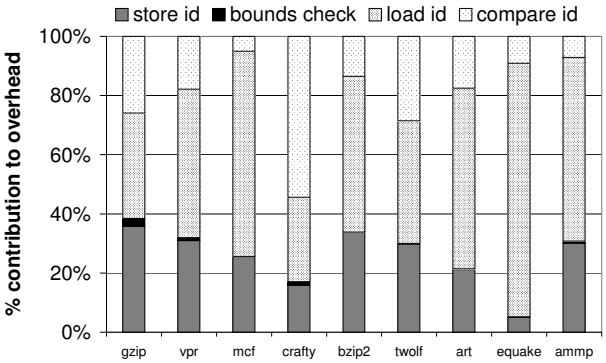

Figure 7: Breakdown of the instrumentation overhead.

time executing in the kernel. For example, it could be applied to Internet browsers or Web servers, which are the target of frequent attacks. In Section 4.1.4, we show that the overhead is significantly lower when using data-flow integrity enforcement to protect a Web server.

It is hard to perform a detailed quantitative comparison with previous techniques due to differences in hardware, operating system, and compiler. But we can use published results obtained using the same benchmarks to put our overhead in perspective. Program Shepherding [27] and CFI [5] have lower overhead but these techniques cannot detect non-control-data attacks. The overhead of either variant of DFI is significantly lower than the overhead incurred by a state-of-the-art C bounds checker: CRED [34] incurs an overhead of nearly 300% in bzip2 and 100% in gzip [34]. The overhead of software implementations of taint checking [31, 16] is also significantly higher, for example, TaintCheck [31] ran bzip2 37.2 times slower than without instrumentation.

We also compared the space overhead introduced by *interproc DFI* relative to *base* during the execution of each program. We compare the peak physical memory usage during execution, as reported by the operating system. The results are shown in Figure 6. As expected the overhead is approximately 50% because we allocate a 2-byte entry in the RDT for each 4-byte word in the instrumented program.

4.1.2 Overhead breakdown

To better understand the sources of overhead, we ran experiments to breakdown the overhead of *interproc DFI* into several components. The experiments removed components from the instrumentation one at a time. Figure 7 shows the breakdown normalized by the total overhead for each program.

The overhead of `SETDEF`s is the sum of the overhead of the first two components: storing the identifier into the RDT and performing a bounds check to protect the RDT from tampering. The remaining overhead is due to `CHECKDEF`s: loading the identifier from the RDT and checking if it is in the set of reaching definitions. A `SETDEF` is on average 3.6 times more expensive than a `CHECKDEF` but the programs execute between 4 times and 25 times more `CHECKDEF`s than `SETDEF`s. Therefore, the total overhead of `CHECKDEF`s is significantly higher than that of `SETDEF`s.

The bulk of the overhead is due to memory accesses to the RDT and the cache pollution they introduce, which is not surprising given the gap between processor speed and memory latency. The overhead of the bounds check on writes is negligible because our optimization can remove many bounds checks. Membership checks are more expensive because we always perform at least one comparison and sometimes more: the average number of comparisons is 1.5. These membership checks account for most of the overhead in crafty because its small working set makes accesses to the RDT relatively less expensive.

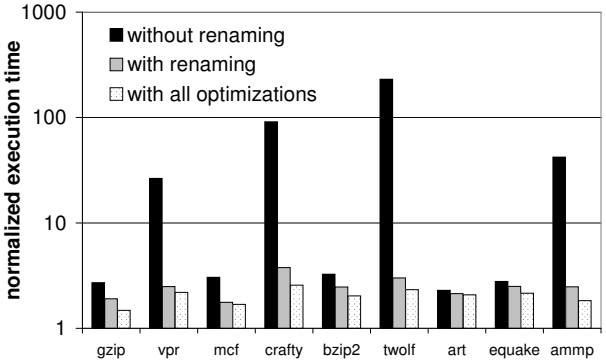

Figure 8: Execution time without optimizations, with the renaming optimization, and with all optimizations (normalized by the execution time without instrumentation).

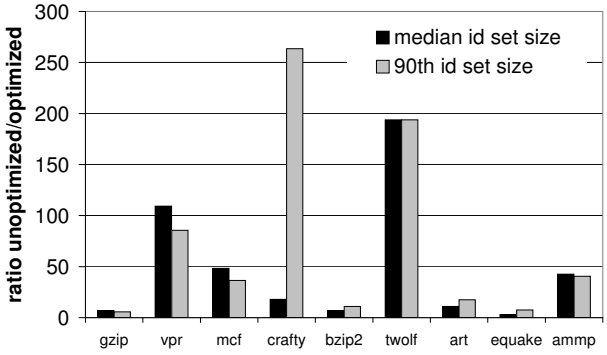

Figure 9: Median and 90-th percentile definition set sizes without the renaming optimization divided by the corresponding values with the optimization.

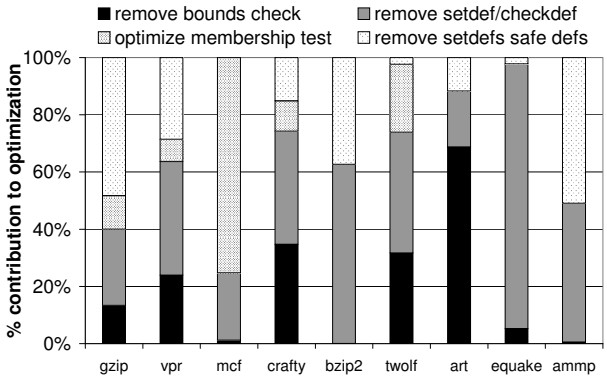

Figure 10: Contribution of each optimization (other than renaming) to the reduction in execution time.

4.1.3 Impact of optimizations

We also measured the overhead reduction afforded by the different optimizations.

The optimization with the most significant impact is the renaming optimization, which assigns the same identifier to all definitions that have the same set of uses. We compared the running time of *interproc DFI* with and without this optimization. The version without the optimization assigns a unique identifier to each instruction that defines a value. Figure 8 shows the execution times without optimizations, with the renaming optimization, and with all the optimizations. The execution times are normalized by the execution time without instrumentation and the y-axis has a logarithmic scale.

The results show that the renaming optimization is fundamental for data-flow integrity enforcement to be practical: some benchmarks are more than an order of magnitude slower without it. The performance difference is due to a large reduction in the size of reaching definition sets. We measured the size of reaching definition sets computed by the interprocedural analysis with and without the optimization. Figure 9 shows the median and 90-th percentile set sizes without the optimization divided by the corresponding values with the optimization. The ratios are very large: they vary between 3 and 264. The renaming optimization brings the median definition set size to 3 or less for 6 of the benchmarks.

We also ran experiments to measure the impact of the other optimizations discussed in Section 3. The experiments added one optimization at a time starting from the version with the renaming optimization. Figure 10 shows the contribution of each optimization to the difference in execution time between the version with the renaming optimization and the version with all optimizations.

The results show that all the optimizations play an important role in at least one benchmark program.

4.1.4 Web server overhead

We also ran the SPEC Web 1999 benchmark to measure the overhead added by our instrumentation to the `NullHttpd` Web server. The server ran on a Dell Precision workstation 350 with a 3GHz Intel Pentium 4 processor and 2GB of RAM. The clients ran on a Dell Latitude D600 laptop with a 2GHz Intel Pentium processor and 1GB of RAM. Both machines ran Windows XP professional with service pack 2 and they were connected by a 100Mbps D-Link Ethernet switch. We configured the clients to request only static content. Since we do not instrument the Perl scripts that handle requests for dynamic content, our overhead is higher with static content. We present the average of three runs.

We measured the average response time and throughput for a base version of the server without instrumentation and for a version of the server instrumented to enforce data-flow integrity. For both versions, we increased the number of clients while the results complied with the benchmark rules. The results stopped complying at 60 clients for the version with instrumentation and at 80 clients for the version without instrumentation. Figure 11

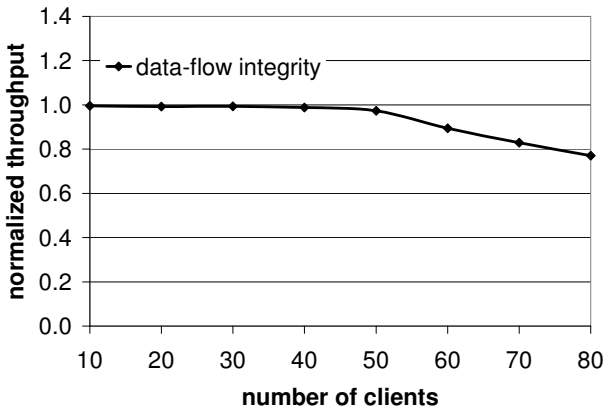

Figure 11: Spec Web throughput with instrumentation normalized by the throughput without instrumentation.

shows the throughput in operations per second for the version with instrumentation normalized by the throughput without instrumentation.

When server load is low, the overhead of our instrumentation is also low because it is masked by other overheads (for example, the time to send requests and replies across the network). The results show that the throughput of both versions of the server is almost identical with up to 40 clients. Additionally, the average operation response time in an unloaded server (10 clients) is only 0.1% longer with instrumentation than without.

When server load is high, the overhead of data-flow integrity enforcement increases because the server is CPU-bound in this benchmark. The overhead of enforcing data-flow integrity increases up to a maximum of 23%.

4.2 Effectiveness against attacks

We used several exploits to evaluate the effectiveness of data-flow integrity enforcement at stopping both control-data and non-control-data attacks. We used a benchmark with synthetic exploits [40] and several exploits of real vulnerabilities in existing programs. This section describes the programs, the vulnerabilities, and the exploits.

4.2.1 Synthetic exploits

We ran the benchmark program that was described in [40]. This benchmark has 18 control-data attacks that exploit buffer overflow vulnerabilities. The attacks are classified according to the technique they use to overwrite control-data, the location of the buffer they overflow, and the control-data they target. There are two techniques to overwrite control-data. The first overflows a buffer until the control-data is overwritten. The second overflows a buffer until a pointer is overwritten, and uses an assignment through the pointer to overwrite the control-data. The attacks can overflow buffers located in the stack or in the data segment, and they can target four types of control-data: the return address on the stack, the old base pointer on the stack, and function pointers and longjmp buffers in the stack or in the data segment.

Table 1 shows that data-flow integrity enforcement can prevent all the attacks in the benchmark. The best techniques evaluated in [40] failed to prevent 50% of the attacks. Some recent techniques can also prevent all attacks in this benchmark, for example, CFI [5] and CRED [34].

4.2.2 Real exploits

The first experiment ran the `NullHttpd` HTTP server. This server has a heap overflow vulnerability that an attacker can exploit by sending HTTP POST requests with a negative content length field [2]. These POST requests cause the server to allocate a heap buffer that is too small to hold the data in the request. While copying the data to the buffer, the server overwrites some data structures that are used by the C library to manage the heap. This vulnerability can be exploited to overwrite arbitrary words in memory.

We attacked `NullHttpd` using the technique described in [14]. The attack works by corrupting the CGI-BIN configuration string. This string identifies a directory holding programs that may be executed while processing HTTP requests. Therefore, by corrupting it, the attacker can force `NullHttpd` to run arbitrary programs. This is a non-control-data attack because the attacker does not subvert the intended control-flow in the server.

Data-flow integrity enforcement detects the attack on the first use of the CGI-BIN configuration string after it has been overwritten. It can do this because the library instructions that manipulate the internal heap data structures store invalid identifiers in the RDT entries for the words they write (as described in Section 2.4). This invalid identifier is, of course, not in the reaching definition set for the uses of the CGI-BIN configuration string.

In addition to preventing the attack, data-flow integrity enforcement found the following bug during the processing of `NullHttpd`'s default configuration file:

```
while ((line[strlen(line)-1]=='\n')
         ||(line[strlen(line)-1]=='\r')) {
  line[strlen(line)-1]='\0';
}
```

If `line` contains the string `"\n"`, the second pass through the loop accesses `line[-1]`. This could lead to memory corruption if `line[-1]` contained the character `'\n'` or `'\r'`.

Attack	Target data structure	Detected?
direct overwrite on stack	parameter function pointer	yes
	parameter longjmp buffer	yes
	return address	yes
	old base pointer	yes
	function pointer	yes
	longjmp buffer	yes
direct overwrite on data segment	function pointer	yes
	longjmp buffer	yes
overwrite through stack pointer	parameter function pointer	yes
	parameter longjmp buffer	yes
	return address	yes
	old base pointer	yes
	function pointer	yes
	longjmp buffer	yes
overwrite through data segment pointer	return address	yes
	old base pointer	yes
	function pointer	yes
	longjmp buffer	yes

Table 1: Synthetic attacks detected by data-flow integrity enforcement.

The second experiment ran `SSH`, which is a secure shell implementation from OpenSSH.org that contains an integer overflow vulnerability [3]. The problem is caused by an assignment of a 32-bit integer to a 16-bit integer. If the lower word of the 32-bit integer is 0, the 16-bit integer is assigned 0. This integer is then passed as an argument to `malloc`, which results in the allocation of a buffer that is too small. This vulnerability can be exploited to write any memory position in the server.

We ported the vulnerable `SSH` implementation to Windows and attacked it using the exploit in [14]. This exploit is similar to the example described in Section 2. It overwrites a variable, which is stored in the stack frame of another function, that records whether the user has been authenticated. The attacker can use the server without being authenticated by setting the variable to 1.

Data-flow integrity enforcement catches the attack on the first use of the authenticated variable after it has been overwritten, because the reaching definitions set for the use of the variable does not contain the identifiers of any definition outside the function where the variable is declared. This attack is detected even when data-flow integrity enforcement instruments only local variables without external definitions (like the version that we called *intraproc DFI* in the previous section).

The third experiment ran the `GHTTP` HTTP server, which has a number of vulnerabilities [1]. We ported `GHTTP` to Windows and exploited a buffer overflow vulnerability in a stack-based buffer to corrupt the return address. Data-flow integrity enforcement catches this common control-data attack because of the instrumentation that we add to check the integrity of return addresses (as described Section 2). The identifier stored by the buffer write to the RDT entry of the return address is guaranteed to be different from the expected value of zero.

The final experiment ran `Stunnel`, which is a generic tunnel for security protocols. It is vulnerable to a format string attack [4] because it passes a string received from the network as a format string to the `vsprintf` function. To attack `Stunnel`, we supplied a format string that causes the return address of the function to be overwritten. This is typical of format string attacks and it is another control-data attack. As in the previous experiment, data-flow integrity enforcement detects the attack when the overwritten return address is about to be used by the return instruction.

5 Related work

Many mechanisms have been proposed to protect programs from attacks. CCured [30] and Cyclone [24] proposed safe dialects of C. They can prevent more attacks than data-flow integrity enforcement because they ensure memory safety for programs written in these dialects. However, they require a significant effort to port C applications to the safe dialects, and they require major changes to the runtime. For example, they replace malloc and free by a garbage collector. A garbage collector makes performance hard to predict and it can introduce a significant overhead, for example, the results presented in [30] show that CCured slows down the bc utility by almost a factor of 10.

Other mechanisms can only defend from attacks that overwrite specific targets, such as return addresses, function pointers or other control data (e.g., [18, 17, 35]). While effective against some attacks, these approaches

Application	Vulnerability	Exploit	Detected?
NullHttpd	heap-based buffer overflow	overwrite cgi-bin configuration data	yes
SSH	integer overflow and heap-based buffer overflow	overwrite authenticated variable	yes
STunnel	format string	overwrite return address	yes
Ghttpd	stack-based buffer overflow	overwrite return address	yes

Table 2: Real attacks detected by data-flow integrity enforcement.

can miss many control-data attacks [40] and they cannot defend against non-control data attacks.

Program Shepherding [27] and Control-Flow Integrity [5] are general mechanisms to ensure that a program does not deviate from its control-flow graph. They analyze the source code to compute a control-flow graph and use binary rewriting to enforce integrity of control-flow at runtime. Control-Flow Integrity provides lower overhead than Program Shepherding and it has been shown formally to catch any deviation from the control-flow graph [6]. However, as pointed out by [14], attacks can succeed without changing the control-flow of the target programs. Neither Control-Flow Integrity nor Program Shepherding can detect non-control-data attacks.

Several systems have proposed broad coverage attack detectors based on dynamic taint analysis [37, 15, 19, 31, 13, 16, 22, 33]. These mechanisms can detect both control-data and non-control-data attacks. They have the advantage of not requiring source code; they work on binaries and do not even require symbol information. However, they can have false positives and proposed implementations have high overhead or require hardware support.

There are several bounds checkers for C (e.g., [26, 36, 25, 34]). The Jones and Kelly [25] bounds checker does not require changes to the pointer format. It instruments pointer arithmetic to ensure that the result and original pointers point to the same object. To find the target object of a pointer, it uses a splay tree that keeps track of the base address and size of heap, stack, and global objects. A pointer can be dereferenced provided it points to a valid object in the splay tree. CRED [34] is similar but provides support for some common uses of out-of-bounds pointers in existing C programs. These bounds checkers can detect both control-data and non-control data attacks but they do not prevent all bounds violations. For example, they cannot prevent attacks that exploit format string vulnerabilities or that overwrite data using a pointer to a dead object whose memory was reused. Additionally, they have high overhead because of accesses to the splay tree. Data-flow integrity enforcement can prevent these types of attack and it has lower overhead.

Several systems use static analysis to detect software vulnerabilities in C and C++ programs (e.g., [38, 11, 12]). These systems have been very successful at finding vulnerabilities in existing software and they have the advantage of detecting problems at compilation time. Like data-flow integrity enforcement, they can miss attacks due to analysis imprecision but, unlike it, they can generate a large number of false positives. Other systems have proposed static techniques to enforce user-specified data confidentiality and integrity policies (e.g., [41]). Our goal is different: we want to enforce a low level safety property that is automatically derived from the source code of programs written in an unsafe language.

Debugging tools like Purify [23] can detect many memory errors in programs. They could be used to detect attacks but they have high overhead.

6 Conclusions

We introduced a simple safety property called data-flow integrity and presented an implementation that instruments programs automatically to enforce this property. Since most attacks must violate data-flow integrity to be successful, data-flow integrity enforcement can protect software from most attacks: it can prevent both control-data and non-control-data attacks. Previously, there was no good defense against non-control-data attacks and they will become common once we deploy defenses against control-data attacks.

Our implementation can be used in practice because it can be applied automatically to C and C++ programs without modifications and it does not have false positives. We described several optimizations that reduce the overhead of the instrumentation. These optimizations make it practical to run data-flow integrity enforcement in many applications.

We believe that it is possible to improve the coverage and performance of data-flow integrity enforcement. There has been recent work on more precise points-to analysis that can scale to large programs [12]. We could leverage this work to improve the precision of our interprocedural analysis.

Acknowledgments

We thank our shepherd Robbert van Renesse, Andrew Myers, Antony Rowstron, Paul Barham, Michael Fetterman, Jon Crowcroft and the anonymous reviewers for helpful comments on this work. We thank John Wilander for sharing his library of attacks with us.

References

[1] GHttpd Log() Function Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/5960.

[2] Null HTTPd Remote Heap Overflow Vulnerability. http://www.securityfocus.com/bid/5774.

[3] SSH CRC-32 Compensation Attack Detector Vulnerability. http://www.securityfocus.com/bid/2347.

[4] STunnel Client Negotiation Protocol Format String Vulnerability. http://www.securityfocus.com/bid/3748.

[5] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow Integrity: Principles, implementations, and applications. In *ACM CCS* (Nov. 2005).

[6] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. A theory of secure control flow. In *ICFEM* (July 2005).

[7] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[8] AMME, W., BRAUN, P., ZEHENDNER, E., AND THOMASSET, F. Data dependence analysis of assembly code. In *PACT* (Oct. 1998).

[9] ANDERSEN, L. Program analysis and specialization for the C programming language. PhD thesis, University of Copenhagen, 1994.

[10] APPEL, A. W. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[11] ASHCRAFT, K., AND ENGLER, D. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *IEEE Symposium on Security and Privacy* (May 2002).

[12] AVOTS, D., DALTON, M., LIVSHITS, V. B., AND LAM, M. S. Improving software security with a C pointer analysis. In *ICSE* (May 2005).

[13] CHEN, S., XU, J., NAKKA, N., KALBARCZYK, Z., AND IYER, R. K. Defeating memory corruption attacks via pointer taintedness detection. In *DSN* (July 2005).

[14] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *USENIX Security Symposium* (July 2005).

[15] COSTA, M., CROWCROFT, J., CASTRO, M., AND ROWSTRON, A. Can we contain Internet worms? In *HotNets* (Nov. 2004).

[16] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: End-to-end containment of Internet worms. In *SOSP* (Oct. 2005).

[17] COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium* (Aug. 2003).

[18] COWAN, C., PU, C., MAIER, D., HINTON, H., WADPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Stackguard: Automatic detection and prevention of buffer-overrun attacks. In *USENIX Security Symposium* (Jan. 1998).

[19] CRANDALL, J. R., AND CHONG, F. T. Minos: Control data attack prevention orthogonal to memory model. In *MICRO-37* (Dec. 2004).

[20] DEBRAY, S. K., MUTH, R., AND WEIPPERT, M. Alias analysis of executable code. In *POPL* (Jan. 1998).

[21] HEINTZE, N., AND TARDIEU, O. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In *PLDI* (June 2001).

[22] HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. Practical taint-based protection using demand emulation. In *EuroSys* (Apr. 2006).

[23] IBM. Purify. http://www-306.ibm.com/software/awdtools/purify.

[24] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference* (June 2002).

[25] JONES, R., AND KELLY, P. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging* (May 1997).

[26] KENDALL, S. Bcc: run–time checking for C programs. In *USENIX Summer Conference* (June 1983).

[27] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. P. Secure execution via program shepherding. In *USENIX Security Symposium* (Aug. 2002).

[28] MCCORKENDALE, B., AND SZOR, P. Code Red Buffer Overflow. *Virus Bulletin* (Sept. 2001).

[29] MICROSOFT. Phoenix compiler framework. http://research.microsoft.com/phoenix/phoenixrdk.aspx.

[30] NECULA, G., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems 27*, 3 (May 2005).

[31] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software. In *NDSS* (Feb. 2005).

[32] ONE, A. Smashing the stack for fun and profit. *Phrack 7*, 49 (Nov. 1996).

[33] PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. Argos: an emulator for fingerprinting zero-day attacks. In *EuroSys* (Apr. 2006).

[34] RUWASE, O., AND LAM, M. A practical dynamic buffer overflow detector. In *NDSS* (Feb. 2004).

[35] SMIRNOV, A., AND CHIUEH, T. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *NDSS* (Feb. 2005).

[36] STEFFEN, J. L. Adding run-time checking to the portable C compiler. *Software - Practice and Experience 22*, 4 (Apr. 1992), 305–306.

[37] SUH, G. E., LEE, J., AND DEVADAS, S. Secure program execution via dynamic information flow tracking. In *ASPLOS XI* (Oct. 2004).

[38] WAGNER, D., FOSTER, J. S., BREWER, E. A., AND AIKEN, A. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS* (Feb. 2000).

[39] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *SOSP* (Dec. 1993).

[40] WILANDER, J., AND KAMKAR, M. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS* (Feb. 2003).

[41] ZHENG, L., CHONG, S., MYERS, A. C., AND ZDANCEWIC, S. Using Replication and Partitioning to Build Secure Distributed Systems. In *IEEE Symposium on Security and Privacy* (May 2003).