# Efficient Fine-Grained Binary Instrumentation with Applications to Taint-Tracking*

Prateek Saxena†, R. Sekar and Varun Puranik
Department of Computer Science
Stony Brook University, Stony Brook, NY, USA.

## ABSTRACT

Fine-grained binary instrumentations, such as those for taint-tracking, have become very popular in computer security due to their applications in exploit detection, sandboxing, malware analysis, etc. However, practical application of taint-tracking has been limited by high performance overheads. For instance, previous software based techniques for taint-tracking on binary code have typically slowed down programs by a factor of 3 or more. In contrast, source-code based techniques have achieved better performance using high level optimizations. Unfortunately, these optimizations are difficult to perform on binaries since much of the high level program structure required by such static analyses is lost during the compilation process. In this paper, we address this challenge by developing static techniques that can recover some of the higher level structure from x86 binaries. Our new static analysis enables effective optimizations, which are applied in the context of taint tracking. As a result, we achieve a substantial reduction in performance overheads as compared to previous works.

## Categories and Subject Descriptors

D.3.4 [**Languages**]: Processors, Optimization

## General Terms

Languages, Security, Performance

## 1. INTRODUCTION

A number of recent advances in software security are based on fine-grained program transformation techniques. Runtime policy enforcement techniques (e.g., program shepherding [16], CFI [1] and XFI [13]), memory error detection techniques (e.g., bounds-checking C [15], Valgrind [29]), exploit-protection techniques (e.g., StackGuard [11] and some randomization techniques [5]) are all examples of such transformation based defenses. More recently, dynamic taint-tracking (also known as information flow tracking) has become very popular due to its applicability for detecting a wide range of attacks [24, 25, 14, 33], malware analysis [32, 12, 35], automated signature generation [10, 24], and so on.

Many security-enhancing transformations maintain and check *metadata* associated with a program's data. For instance, memory error detection requires instrumentation to maintain metadata about memory allocations, pointers and arrays used in a program. Similarly, taint-tracking requires every data move (or arithmetic) operation to be instrumented to compute the taint of the destination operand. Architectural modifications (for metadata maintenance) can support efficient fine-grained instrumentation [31, 8], but are not as flexible or practical as software-based implementations. Consequently, most practical implementations rely on software-based approaches for fine-grained instrumentation.

Since source code is often unavailable for COTS software, binary (rather than source-code) instrumentation is preferable. Moreover, binary instrumentation is applicable to code written in many languages, including C, C++, and assembly[1]. Unfortunately, fine-grained binary instrumentation incurs significant performance penalties. For instance, some of the earlier works on taint-tracking slowed down programs more than ten times, while the best known performance overheads come in at above 300% [26]. In contrast, source-code transformations have achieved much lower overheads (in the range of 50%) [33, 17] due to the following factors.

- Source-code instrumentation techniques can "piggy-back" on powerful optimizations provided by the compiler for the source language. In contrast, static analysis and optimizations are much harder on binaries due to the lack of high-level information such as variable and function boundaries, data types, etc.
- Robustness requirements of fine-grained transformations have so far driven the use of dynamic translation [16, 20, 23, 6] for handling large COTS binaries, as opposed to static binary rewriting. Since these techniques operate entirely at runtime, they are constrained to use very light-weight analysis techniques. Moreover, since they transform each basic block during its first execution, it is difficult to perform static analyses across basic blocks, thereby further limiting the nature and effectiveness of optimization techniques that can be used.

We bridge this performance gap in this paper by developing effective yet scalable static analysis techniques for binaries. We then develop static binary instrumentation techniques that achieve substantial improvements in performance as compared to the best results reported previously [26]. Although our implementation targets taint-tracking, our analysis techniques are more broadly applicable. Below, we describe the challenges of efficient binary instrumentation and summarize the contributions of this paper.

*This research is supported in part by an ONR grant N000140710928 and an NSF grant CNS-0627687.
†This author is currently at University of California, Berkeley.


*CGO'08,* April 5–10, 2008, Boston, Massachusetts, USA.


[1]Many popular applications such as Mozilla Firefox, GIMP, and several media players make significant use of assembly code.

```
typedef struct {int a;} ABC;
ABC array[10], min = {0};

int subtract (int a, int b) {
  return (a - b);
}

int (*comp)(int, int)
    = subtract;

ABC get_min(ABC* x) {
  ABC temp = min;
  if (less_than(x, &min))
    temp = *x;
  return temp;
}

int less_than(ABC *x, *y) {
  return comp(x->a, y->a);
}

void main () {
  int i = 0, size = 10;
  for (i=1; i<size; i++)
    min=get_min(&array[i]);
}
```

```
<get_pc_thunk_bx>:
  mov [esp], ebx
  ret

<get_min>:
  sub 28, esp
  mov ebx, 12[esp]
  call get_pc_thunk_bx
  add STATIC_OFF_2,
      ebx
      .....
  mov ebp, 24[esp]
  mov 32[esp], ebp
      .....
  call less_than
      .....
  mov esi, [ebp]
  mov ebp, eax
  mov 24[esp], ebp
  add 28, esp
  ret 4
```

```
<subtract>:
  mov 4[esp], eax
  sub 8[esp], eax
  ret

<less_than>:
  push ebx
  sub  8, esp
  mov  20[esp], eax
  call get_pc_thunk_bx
  add  STATIC_OFF_1,
       ebx
  mov  [eax], eax
  mov  eax, 4[esp]
  mov  16[esp], eax
  mov  [eax], eax
  mov  eax, [esp]
  mov  off_funcptr[ebx],
       eax
  call [eax]
  add  8, esp
  pop  ebx
  ret
```

```
<main>:
   lea  4[esp], ecx
   and  -16, esp
   push -4[ecx]
   push ebp, edi, esi, ebx
        .....
   lea  20[esp], edx
Z: mov  edx, 16[esp]
   mov  array_offset[ebx], eax
   mov  min_offset[ebx], ebp
   lea  4[eax], esi
   lea  40[eax], edi
L: mov  16[esp], eax
X: mov  esi, 4[esp]
   add  4, esi
Y: mov  eax, [esp]
   call get_min
   sub  4, esp
   cmp  edi, esi
        .....
   jne  L
   pop  ecx, ebx, esi, edi, ebp
   lea  -4[ecx], esp
   ret
```

**Figure 1: A simple C program that finds the minimum element in an array, and its compiled code.**

### *Challenges in Efficient Binary Instrumentation*

Consider the C-program shown in Figure 1 that finds the smallest element in an array `array`, and its compiled code obtained using `gcc-4.1 -O2 -fPIC -fomit-frame-pointer`. This example has been designed to capture the features of large C/C++ programs, including indirect calls, parameter passing by reference, etc. For comparing `array` elements, this program uses the `less_than` function, which in turn invokes another function `subtract` using a function pointer `comp`. The assembly code in this example has been pruned for conciseness, and certain instructions are grouped together into single "psuedo-instructions" to improve readability. This example illustrates the following challenges of binary static analysis and instrumentation:

- *Missing information on function boundaries and stack conventions.* Functions in binaries may have multiple entry and/or exit points, and may exit as a result of a jump rather than a return (usually due to tail-call optimization). They may not follow typical conventions regarding the stack, and may use the stack pointer and base pointer registers (ESP and EBP respectively on the x86 architecture) in non-standard ways. To cope with these difficulties posed by large code bases, compiler variations and compile-time options, our approach strives to minimize assumptions about the compiler or the binary being transformed. Specific assumptions made include: the stack grows down, the stack pointer is preserved by any function that may be called indirectly, and that a procedure does not access the activation record of other functions unless pointers to those records were passed into the function. We don't make assumptions regarding parameter passing (i.e., they may be passed on the stack, via registers or through global memory), caller- and callee-save registers, use of base pointers, multi-threading, etc.
- *Position-independent code (PIC).* Note that a function `get_pc_thunk_bx` is called that returns (in the `ebx` register) the location of the instruction that invoked it. The base address for all static variables is computed by adding an offset (`STATIC_OFF_2`) to this address. Since binary rewriting involves relocating functions without relocating static data, it is necessary to recognize and "fix up" PC-relative data accesses, or avoid relocating the code fragments that call `get_pc_thunk_bx`. The main difficulty here is that different compilers (or compiler versions) use different mechanisms for PC-relative addressing. Rather than relying on compiler-specific idioms, we use our static analysis to detect PC-relative data accesses.
- *Scalability and modularity.* In order to cope with large programs, it is necessary to develop modular static analysis and rewriting techniques that scale to large programs, while being fast and accurate enough.
- *Missing information about local variables.* Although most local variable accesses use `ESP` as a base pointer, some functions (e.g., `main`) use other registers such as `ECX`. Similarly, `EBP` is typically used to access parameters, but in optimized code such as the above example, `EBP` may be used as a general-purpose register. Rather than using unreliable heuristics, we perform a static analysis that accurately identifies local variable accesses.
- *Missing information about actual parameters.* One may expect that actual parameters would be explicitly pushed on the stack, but as shown by the instructions `X` and `Y` in `main`, they may be stored using stack-relative addresses. Worse, in optimized code, the actual parameters may be in temporaries that happen to be at the top of the stack. Thus, a local examination of the call-site won't reveal the number of actual parameters. Hence we rely on a static analysis of the callee to determine the parameters that are passed on the stack, or via registers. Except for indirect calls, we do not rely on compliance with an application-binary interface (ABI) regarding the use of registers, as it may not be observed for intra-module calls.

- *Aliasing.* In the presence of indirect memory accesses, the soundness of many optimizations requires an analysis of possible values held by a pointer. For instance, it is safe to store the metadata associated with a memory location $l$ in a register $R$ if you can statically identify all the instructions that will access $l$, and instrument them so that they access the metadata from $R$. In the presence of aliasing, it is possible that multiple pointer expressions may reference the same memory, making it difficult to identify all such instructions.

  Accurate pointer analysis is a challenging problem even on source code, and is made even more difficult for binary code due to the absence of high-level information such as variables, array sizes, types, etc. Our approach for pointer analysis is guided by the empirical observation that on register-starved, stack-oriented architectures such as x86, accesses to local memory (i.e., locations on the activation record for a function) significantly outnumber accesses to global or heap memory [5]. Moreover, most of these memory locations are accessed within just a single procedure, making it easier to accurately reason about these accesses. In contrast, accesses to global and heap-allocated variables tends to be more distributed spatially (across several procedures) and temporally (i.e., multiple function calls may be made and/or many possible program paths traversed between two accesses to the same memory), which makes it difficult to reason about them accurately. Our approach hence strives to reason accurately about most local variable and parameter accesses, while treating global and heap accesses conservatively.
- *Functions with (unusual) side-effects.* Note that `get_min` deallocates the return structure pointer passed into it (at instruction Y in `main`), rather than expecting its caller to perform the deallocation. As a result `ESP` value is not preserved across the call to `get_min`. Once again, we rely on a static analysis to cope with these side-effects.

### Contributions

We make the following contributions in this paper.

- We develop a flow-sensitive static analysis technique in Section 4 that we call as *stack analysis* for recovering and reasoning about the values stored in registers and variables in the activation record of each function. Although it shares some of the same objectives as the value-set analysis (VSA) [2], it differs in several ways. First, our focus is on a modular and fast analysis that ignores global and heap memory, whereas VSA trades off speed for increased accuracy in tracking global and heap memory. Second, by using a different abstract domain, we are not only able to derive that the contents of certain memory locations (or registers) aren't equal, which is useful for ruling out aliases, but also that some values are equal or that they differ by a constant. Reasoning about equalities is important for minimizing assumptions on the way ESP and EBP are used, and to assert certain properties such as ensuring that callee-saved registers are left unchanged across function activations.
- Based on stack analysis, we develop an *escape analysis* on binaries in Section 5 that detects instances when a reference to a local variable escapes a function. This can happen because the address is passed to another function (via registers or through procedure parameters), or is stored in non-local memory. In our experiments, we found that most functions are "safe," i.e., local addresses of these functions do not escape them. Hence we are able to realize most benefits of our optimization even though they mainly target such safe functions.
- In Section 6, we present three sound optimizations that improve the performance of metadata operations:
  - *Tag-sharing.* We have developed a new static analysis for optimizing taint-computation. It is based on the observation that due to the introduction of temporaries for expression evaluation, and the need imposed by a register-starved architecture (such as x86) for saving registers on the stack, it is common for multiple local memory locations and registers to have the same taint value. Our analysis identifies such locations, and uses a single taint tag for all of them. As a result, tag updates can be eliminated for data movement operations involving variables that share the same taint tag. Note that this optimization is applicable to binary as well as source-code instrumentation approaches.
  - *Metadata-caching.* First, we store metadata for registers (and a subset of local variables) in one or more general purpose registers. Second, we use a shadow stack for holding metadata for stack-memory. Although this technique does not reduce memory accesses, it improves performance since metadata accesses can use EBP or ESP as the base register instead of having to find another free register, which would in turn require a save/restore to memory.
  - *Code specialization.* We have explored the generation of two versions of code for each function. The first version is used when all local variables and registers are untainted, and the other when some of them are tainted. In the first version, tag updates can be avoided for all register and local memory computations. Memory store operations require writing a zero to the associated taint storage. Memory loads require a check to determine if the source location is tainted, and if so, jump to the second code version. Although conceptually similar to the *fastpath* optimization outlined in [26], our technique is more effective due to its use of more powerful analysis.
- We have developed an implementation that is robust enough to handle moderately large programs in C and C++ such as Gaim (234 KLOC) and GIMP (742 KLOC) on Linux. It can handle multi-threaded programs. It is capable of defending itself from typical software exploits such as memory corruption attacks, but our optimizations assume benign code, i.e., code that does not actively evade our defenses. Our performance evaluation, carried out on a subset of SPEC95 INT programs, shows substantive speedups over previous techniques.

### Paper Organization

We begin with background on taint-tracking and static binary rewriting in Section 2. Following this, we present our static analyses in Sections 3 through 5, and then describe the optimizations based on these analyses in Section 6. Experimental evaluation is described in Section 7, followed by a discussion of related work in Section 8 and concluding remarks in Section 9.

## 2. BACKGROUND

### 2.1 Dynamic Taint-Tracking

Dynamic taint-tracking associates one or more bits of *taint* with each byte (or word) of program data. When working with low-level languages such as C or binaries, a global array *tag*, indexed by memory locations, is used to maintain these taint tags. Specifically, $tag[l]$ stores the taint associated with the data stored in memory location $l$.

Taint originates at *source* functions, such as network or file read operations. The idea is to mark data as tainted if it is received from an untrustworthy source. In the simplest case, one bit of taint is sufficient, but there are situations where multiple bits are useful, e.g., to distinguish between multiple input sources or to distinguish between trust levels. Attacks are detected by checking the taint bits at *sink* points. For instance, control-hijack attacks involve tainted data being used as a code pointer. More generally, user-specified *policies* may be enforced regarding the use of tainted data at these sinks, and a wide range of attacks detected as violations of these policies [33].

Taint-tracking requires a program transformation to propagate taint from source operands to destination operands of every instruction. Constants are considered as untainted, while the result of any arithmetic, logical, or data movement operation is deemed tainted if any of the input operands is tainted. While much of the research on information flow concerns itself with control dependences and implicit flows, most works that employ dynamic taint-tracking for attack detection [31, 24, 33, 26] are focused mainly on data dependences in order to minimize false positives.

Some of the dynamic taint-tracking approaches have relied on architectural modifications for taint instrumentation [31]. Taintcheck [24] uses dynamic translation (using Valgrind [23] or DynamoRIO [16]). For reasons mentioned earlier, binary instrumentation without optimization leads to heavy performance overheads, typically slowing programs down by a factor of 10 or more. Several effective optimizations have been suggested in [26, 9], such as minimizing register save/restores, use of faster instructions, and so on. These optimizations serve as our starting point. We combine them with several powerful static analysis techniques that can be applied within our static rewriting framework so as to achieve further significant improvements. To conserve space, we have not described the basic taint transformation or the optimizations that we have borrowed from [26], referring the reader to that paper for details.

### 2.2 Static Binary Rewriting

Static binary rewriting operates offline on executables and libraries [18, 34, 30], while dynamic binary rewriting techniques operate on code stored within the memory of a running process [16, 20, 6, 28]. The two differ primarily in that offline techniques need deeper knowledge of executable formats such as ELF or PE, and can use more expensive analysis or instrumentation techniques. We use an offline approach that uses code derived from LEEL [34] for ELF related editing, and a backend based on nasm assembler for generating machine code for new instructions introduced during rewriting.

Two important issues in static binary rewriting, as opposed to dynamic translation, are as follows. First, static rewriting requires the entire program to be statically disassembled. Several disassembly techniques have been developed [27], but the problem of accurate static disassembly of all "stripped" binaries (i.e., binaries that contain no symbol information) isn't fully solved yet. Since our focus is not on disassembly, we have simplified our implementation task by assuming that information about function entry points is included in the binary. With continued advances in disassembly techniques, it should be possible to eliminate this assumption from our implementation[2]. We point out that unlike the Vulcan [30], we *do not* assume the availability of any high level information regarding variables, types, etc.

Second issue in static binary rewriting is that in-place instrumentation is generally not possible since instrumented code usually requires more space than the original code. Simply moving functions to a new location that has the required space isn't feasible either. Such moves requires all the calls to these functions to be redirected to their original location, but this is difficult in the presence of indirect calls whose destinations cannot be statically determined. In general, without additional compiler-provided information such as relocation information, there is no safe way to relocate functions. We employ the common technique of introducing jumps from the original function entry points to the corresponding entry points in the instrumented version [21]. The rest of the original code is replaced with an invalid opcode so that any jumps into that code results in a runtime exception. This is done so that (a) implementation bugs that result in such jumps would be identified and fixed, and (b) attempts to evade security checks by executing uninstrumented code version will be caught. To deal with instances where insufficient space is available at instrumentation points (to hold 5 byte x86 jump instructions), we use two-level jumps — a first near-jump to an intermediate code block that performs the final dispatch to the target code.

## 3. IDENTIFYING FUNCTIONS

As a first step in instrumentation, our technique computes *basic blocks* that define the basic unit for rewriting. A basic block is a sequence of instructions that has a single entry point and a single exit point. Basic blocks are disjoint. Due to rewriting, basic blocks may expand, and hence jumps to basic blocks need to be fixed up. No adjustments are necessary within a basic block as code executes sequentially.

Given our goal of performing static analysis at the procedural level, we need to recover a notion of functions in binary code. We define an *assembly function* as a collection of basic blocks that has a single entry point that is reached using a *call* instruction located outside the function; and one or more exit points that transfer control from the function to some code outside. Note that the entry point may be reached using *jump* instructions from outside the assembly function, but there has to be at least one *call* to the entry point. An exit point may be a *return* instruction, or a jump to the entry point of another assembly function. Note that this definition does not permit functions with multiple entry points. When they do arise, we treat it as multiple disjoint assembly functions by creating a unique copy of shared basic blocks (i.e., basic blocks that may be reached from more than a single entry point) for each assembly function.

[2]For instance, our approach can work well with robust disassembly techniques that rely on a hybrid approach, such as BIRD [21], which perform static disassembly of most code, while relying on runtime disassembly and instrumentation for the rest of the code.

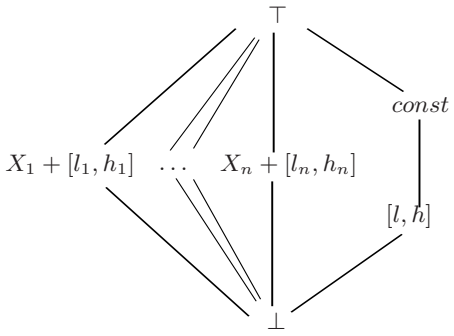


**Figure 2: The abstract domain for stack analysis.**

| Instruction ($I$) | Abstract store ($\mathcal{A}'$) |
|---|---|
| $R := c$ | $Upd(\mathcal{A}, [R \mapsto [c, c]])$ |
| $R := R'$ | $Upd(\mathcal{A}, [R \mapsto \mathcal{A}[R']])$ |
| $R := *(R')$ | $Upd(\mathcal{A}, [R \mapsto \bigcup_{x \in \mathcal{A}[R']} \mathcal{A}[x]])$ |
| $*(R) := R'$ | $Upd(\mathcal{A}, [x \mapsto \mathcal{A}[R']])$, if $\mathcal{A}[R] = \{x\}$<br>$Upd(\cdots(Upd(\mathcal{A}, [x_1 \mapsto \mathcal{A}[x_1] \cup \mathcal{A}[R']]) \cdots$<br>$\cdots), [x_n \mapsto \mathcal{A}[x_n] \cup \mathcal{A}[R']])$<br>if $\mathcal{A}[R] = \{x_1, ..., x_n\}, n > 1$ |
| $R := R_1 + R_2$ | $Upd(\mathcal{A}, [R \mapsto \bigcup_{r_1 \in \mathcal{A}[R_1], r_2 \in \mathcal{A}[R_2]} r_1 \oplus r_2])$ |
| $call(f)$ | $Upd(\cdots(Upd(\mathcal{A}, [x_1 \mapsto applysum(\mathcal{A}[x_1], f, x_1)]) \cdots$<br>$), [x_n \mapsto applysum(\mathcal{A}[x_n], f, x_n)])$<br>where $ModifiedNonLocals(f) = \{x_1, ..., x_n\}$ |

**Figure 3: Abstract interpretation for stack analysis.**

Our assembly function abstraction incorporates a notion of call stack that holds return addresses, and the notion that the `esp` register points to the top of the stack, and that the stack grows down. But it does not assume a stack-based model for parameter passing. Except for indirect calls, it does not make assumptions on caller-save or callee-save registers. These features enable our technique to handle compiler-generated as well as hand-written assembly code, including some low-level code found in libraries such as `glibc`.

## 4. STACK ANALYSIS

Stack analysis is aimed at deriving static estimates of the values of registers and local memory, i.e., memory locations in the activation record of (assembly) functions. Note that the notion of an activation record is closely related to the value of the `esp` register at the function entry point: it refers to the region of memory surrounding this `esp` value that are accessed by the function.

We use an abstract interpretation over the domain shown in Figure 2. Points in this domain take the form $Base + [l, h]$, where $Base$ and $h$ are optional. $Base$, if present, is a symbolic value $X$ that denotes the value of a specific register or local memory at the entry point of a function. A unique symbolic value is associated with each distinct register and each local memory location. A missing base is treated as equivalent to zero, and a missing $h$ is treated as equal to $l$. Both $l$ and $h$ are (possibly negative) integers. Note that the symbolic value corresponding to the initial `esp` value, denoted $BaseSP$, plays a special role since it defines the notion of local memory.

If a register (or a local variable) has an abstract value $X + [l, h]$ at a program point, that means that its concrete value will be in the range of $X+l$ to $X+h$ (inclusive). The abstract value $[l, h]$ denotes a concrete value between $l$ and $h$. The abstract value *const* denotes an unknown concrete value, with one limitation: if this value is used as an address, then it does not point to local memory addresses. This captures the assumption that addresses of local variables are "created" by an activation of a function, and pointers to those variables cannot exist prior to this. (As described in the next section, an *escape analysis* is used to determine instances when a local variable address is stored in a global memory. In those cases, global memory reads will return $\top$ rather than *const*.)

In order to simplify our description, we describe stack analysis in Figure 3 using a simplified language that resembles a RISC instruction set. The definition in Figure 3 specifies the abstract store $\mathcal{A}'$ that results from the execution of an instruction $I$ with an abstract store $\mathcal{A}$. In the figure, $R$ (possibly with subscripts) refers to a register. The abstract store associates a set of up to $k$ abstract values (where $k$ is a small constant) with each register and memory location. The abstract store distinguishes between different locations in local memory, but does not do so among global or heap memory locations. The initial content of global and heap memory locations is given by *const*. The notation $\mathcal{A}[l]$ denotes the contents of the abstract store at location $l$, while the function $Upd$ is used to update its contents. Note that there are two cases for memory updates. If the location being updated is precisely known then its abstract value can be replaced. If it is not known precisely, then we cannot update the location; instead, we conservatively assume that the new value of each of these locations should include the value of the right-hand side of the assignment. Furthermore, recall that $\mathcal{A}$ only distinguishes between different locations in the local memory, i.e., it distinguishes between $BaseSP + x$ and $BaseSP + y$ when $x \neq y$. All addresses that are not of this form are treated as if they are a single abstract location.

For an arithmetic operation '+', the corresponding abstract operation is denoted by '$\oplus$'. For two abstract values $a$ and $b$, if one of them is of the form $X + [l_1, h_1]$ and the other is of the form $[l_2, h_2]$ then $a \oplus b$ is given by $X + [l_1 + l_2, h_1 + h_2]$. If they are of the form $X + [l_1, h_1]$ and $Y + [l_2, h_2]$ then the result is *const* if neither $X$ nor $Y$ is $BaseSP$, and is $\top$ otherwise. If an instruction causes the number of abstract values associated a location to exceed $k$, an appropriate generalization is used to reduce the number of values to $k$ or less.

Conditional branches and merges are handled in the usual way. In particular, the abstract store at a merge point is set to be the union of abstract store values at the end of each basic block that transfers control to the merge point. Although we could restrict the abstract store to contents that is consistent with the condition guarding a branch, we have not done this currently for simplicity.

For handling call-returns, our analysis computes summaries for each called function and applies these summaries at the calling point. Note that there is a single summary, regardless of the calling context. A summary for a function $f$ captures the following information:

- *change in* `esp` as a result of invoking $f$. This value will typically be zero, although it can also be $\top$ or non-zero.
- *maximum size of activation record of* $f$, which captures the range $[BaseSP + l, BaseSP + h]$ of local addresses

ever accessed by $f$, where $BaseSP$ denotes the value of `esp` register at the entry of $f$, and $l, h \in [-\infty \cdots \infty]$.

- *input parameters to* $f$, which are registers or local variables of $f$'s caller that are possibly used before being clobbered in $f$.
- *changes to registers and parameters of* $f$ as a result of $f$'s execution. Note that our abstract interpretation captures the values of registers and local memory of $f$ in terms of their values at the entry point of $f$. Thus the abstract value of registers and parameters at the exit point of $f$ provides the summary we seek.

Given these summaries, our abstract interpretation uses a function *applysum* to update the abstract store to reflect local memory and register changes specified in the summary. If the summary for a function $f$ indicates that it may leave a location $l$ unmodified or change it to one of the values $x_1, ..., x_n$ then *applysum* associates the set of abstract values $\mathcal{A}[l] \cup \{x_1, ..., x_n\}$ with the location $l$.

Note that the summary information we aim to compute depends on the effect of the functions that are called by $f$. For example, if a function changes the value of `ESP` by $k$, it has an effect in its caller at the point of the call. Therefore, the analysis uses multiple passes. It starts with the base case that called functions leave the `ESP` value unchanged, and access no parameters. In the first pass, each function is analyzed separately and a summary set representing the first approximation of its effect is generated. In subsequent passes, this summary information refines the abstract computation at function call points. As a result, better approximations of summary sets are produced after each pass, until a fixed point is reached. The analysis is modular and works well in practice.

Typically, there is a small subset of the x86 instructions that our analysis needs to deal with, since address values are not involved with most instructions. In case of loops, we may encounter successive approximations that are elements of an infinitely ascending chain, such as when incrementing addresses or integers in a loop. In such cases, we must perform widening [7] in order to ensure termination of stack analysis. Our current implementation resorts to a very simple form of widening that widens the integer interval component of an abstract value to $[-\infty, 0]$, $[0, \infty]$ or $[-\infty, \infty]$ after inspecting the abstract value for the first two iterations of a loop. This simplifies our implementation but may lead to some imprecision when dealing with arrays. So far, we have not implemented a more sophisticated widening strategy since it is typically difficult in binaries to reason accurately about array accesses within a loop.

Our analysis assumes that programs follow a "standard" compilation model, i.e., the stack grows downwards, and the return address is pushed on the stack, and this is followed by the callee's function activation. For those calls where our analysis is unable to accurately compute the effect on `EBP` and `ESP`, e.g., indirect calls, it assumes that these registers are left unchanged by the callee (as per the ABI).

# 5. ALIASING AND ESCAPE ANALYSIS

Any sound analysis that reasons about the values of memory-resident objects has to deal with the possible effects of aliasing. Although our stack analysis typically provides enough information to rule out aliasing of most local variables within a procedure, a global analysis is needed to rule out aliases across the entire program. Such global analysis is typically expensive, and moreover, is complicated by features common in real-world programs such as indirect calls (where the target function is unknown), C++ exception handling, signals, and the need to accurately reason about the contents of global memory. We have therefore chosen an alternative technique that reasons about aliasing among local variables, while conservatively assuming that global references could be aliased.

Recall that during the stack analysis, we do not maintain any information about the contents of global memory. As a result, if a pointer value is read from global memory, we cannot rule out the possibility that this pointer may reference local memory. Similarly, a function may create aliases to its local memory by passing a pointer to some of it local variables to another function. Our technique copes with these possibilities using a simple analysis that reasons whether pointers to local memory of a function may "escape" the function, i.e., be potentially accessible in registers, global memory, or the local memory of another function. Since a function's local memory is instantiated at the time of its invocation, we assume that pointers to this memory cannot initially be present anywhere except the stack pointer register. Thus, in order for any register or memory location to contain pointers to local memory, it must propagate from the stack pointer. This can happen when the address of a local variable is explicitly stored in global memory, or is passed as a parameter to another function. Note that in the case of direct function calls, our stack analysis can identify parameters passed through registers or on the stack, and hence can accurately infer if a local variable escapes through these parameters. In case of indirect calls, or in the case of calls to variable argument functions, our stack analysis cannot determine the number of parameters being passed in, so it conservatively assumes that a local variable may escape if any register or local variable contains a pointer to another local variable at the point of this call.

For optimization purposes, we limit our goal to finding the set of all *unsafe* functions, i.e., functions that possibly have any of its local memory accessible through indirect memory references whose targets aren't accurately tracked by stack analysis. Functions in which local addresses escape are clearly unsafe. In addition, since we don't accurately reason about array bounds on the stack, any function that contains arrays on the stack, or modifies its ESP by a non-constant value (e.g., uses "alloca") is deemed unsafe. All other functions are considered *safe*.

# 6. OPTIMIZATIONS

Due to space limitations, we omit a description of some of the standard optimizations used in our code such as register liveness analysis and elimination of dead code. We also omit a description of some of the low-level optimizations, such as those involving instruction selection, which can be found in [26]. Instead, our focus in this section is on new optimizations aimed at speeding up metadata operations, with particular emphasis on taint-tracking.

## 6.1 Metadata Caching

In this section, we describe two ways to speed up accesses to metadata.

**Use of dedicated metadata stack.**

We split the metadata store into two regions: a metadata stack that stores metadata for stack locations, and a global store for storing all other metadata. Actually, there is one metadata-stack for each thread stack. We use the technique suggested in [36] for efficient access to metadata stack: by allocating it at a fixed offset from the stack, we can access metadata stack using a fixed offset from `ebp` or `esp`, thereby avoiding the need for another register to hold the metadata location. Since local variable accesses far outnumber global or heap memory accesses in most programs [33], this optimization yields significant performance improvement in practice.

Note that, for unsafe functions, it is possible that some accesses to local variables may not be statically identified, e.g., if the address of a local variable escapes to a global variable, and is subsequently accessed indirectly using this global variable. One possibility is to refrain from using metadata stack for unsafe functions. We prefer an alternative solution: we use a special metadata value in the global metadata area region corresponding to stack memory. An indirect access will first lookup the metadata value in this global metadata store, and if it has this special value, then a second lookup in the metadata stack is performed. Since such indirect accesses to stack memory are relatively infrequent, this two-step process does not degrade performance significantly. At the same time, significant benefits are gained by eliminating the need for runtime address computation for metadata access for local variables.

We also note that for safe functions, statically unidentified indirect accesses to local memory should not be possible. Hence we mark the corresponding region of global metadata store to indicate that any metadata access using this kind of indirect reference is an error. This test results in the identification memory errors such as those involved in stack-smashing attacks.

**Caching metadata in registers.**

Metadata operations would be further speeded up if metadata can be stored in registers. This is feasible for metadata that takes very few bits, such as taint. In our implementation, we have used two general-purpose registers that provide 64-bits of cache. Note that it is safe to store the metadata associated with a local memory reference to the register cache only if this reference cannot be involved in aliasing.

We use a register assignment algorithm to determine which local memory locations and registers should have their metadata saved in the register cache. Note that at the point of function calls, any metadata corresponding to the actual parameters should be flushed to the metadata stack from the register cache. For indirect calls (or calls to variable argument functions), the entire register cache must be flushed to the metadata stack.

## 6.2 Tag Sharing

On a register-starved architecture such as x86, the same data may reside in multiple locations at different times. It may be moved into a register from local memory to improve performance, or because the instruction set requires a register operand. Subsequently, the register may need to be pushed on the stack so that it may be used to store some other value. All these moves may introduce corresponding moves on the metadata, which can contribute to significant overhead. Note that these metadata moves can be avoided *if we can statically infer that all these copies can share a single copy of metadata.* Below, we describe a single analysis technique that can support both these optimizations in the context of taint-tracking.

Based on our ability to statically analyze the whole program flow graph for a function, we perform standard analyses on the instrumented program. We first represent the program in SSA form, giving a new tag variable whenever we cannot precisely identify which tag variable is accessed using our static analysis and at $\phi$ nodes. Then, treating taint for constants as "0", we can perform an analysis similar to constant propagation. Using this analysis, it is possible to identify a set of variables that require no dynamic taint tracking — such as the variable `i` in `main` of Figure 1 — which is only involved in arithmetic with constants. Further, we perform a similar flow-sensitive common subexpression elimination to determine that many SSA operands share the same tag variables. For example in Figure 1, just before instruction at `L` in function `main`, `esi`, `edi` and `eax` can share the same tag variable.

As a final optimization, we perform liveness analysis for taint variables which is followed by dead code elimination for taint operations. This removes much of the taint processing for the `push` and `pop` instructions in `main` of Figure 1. All of this has an additional impact on register pressure and cache performance.

It should be clear that the tag-sharing optimization is sound for local variables of safe functions: since local memory references aren't aliased, we are free to store their metadata anywhere. For non-local memory references and for unsafe functions, we fall back to storing metadata at a location determined by the address of the data. It is possible to relax this limitation to safe functions so as to include a subset of unsafe functions: specifically, for those functions that are marked unsafe because they make indirect calls (or call variable argument functions), tag sharing can be applied, provided the tag values are copied into memory locations before the call, and restored back after the call.

## 6.3 Code Specialization

This optimization is particularly effective in the context of taint-tracking, where it has been observed [26] that most metadata operations end up propagating a taint value of zero (i.e., the data is untainted). To utilize this bias, we develop two versions of the trusted code: a *fastpath* version that operates when all the registers are untainted, and a *slowpath* version that propagates taint as normal in the presence of tainted registers.

The *fastpath* is considerably faster than the *slowpath*, as it requires no taint computation or propagation for registers (as their taint will be zero), and only a single write operation for clearing the memory taint for store operations. Memory load operations are the only possible way any register could get tainted; hence such instructions require a jump to the *slowpath* version if the loaded data is tainted.

Although conceptually similar to the fastpath optimization developed in [26], the specifics are quite different due to our use of deeper static analysis, and the differences in the way we have defined our fastpath optimization. For instance, their fastpath optimization requires runtime checks at the beginning of each basic block, whereas in our design, checks are needed only for memory loads. Moreover, since

| Program name | Instructions (K) | Analysis Time (seconds) |
|---|---|---|
| ftp | 10 | 6.6 |
| xmms | 74 | 61 |
| VLC player | 167 | 143 |
| httpd | 80 | 66 |
| gaim | 151 | 160 |
| gimp-2.2 | 545 | 269 |
| pdftops | 135 | 321 |
| KPDF | 277 | 511 |

**Figure 4: Analysis time.**

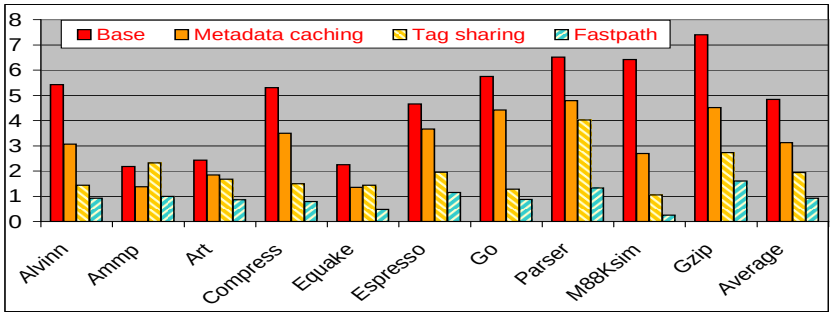


**Figure 5: Overhead for taint-tracking with different optimizations, expressed as a ratio of execution time for uninstrumented code.**

their fastpath does not perform any tag updates, it cannot be applied in some instances where all input operands are untainted. Specifically, if some output operand is tainted, their fastpath technique cannot be used.

#### 6.3.1 *Fastpath for Local Variables*

This optimization is an extension of the fastpath technique to the subset of local variables whose taint is stored in the register cache. We generate two versions of code as outlined earlier — the slowpath version remains unchanged, but the fastpath version makes the additional assumption that the entire register cache used for storing taint is zero. The fastpath code can now free up the register cache for use by the program. Moreover, loads and stores from a local memory location do not need taint check as long as its associated taint is stored in the register cache.

The control switches from fastpath to slowpath if tainted data is loaded from non-local memory. Switching back from slowpath to fastpath is also fast, as it only requires checking of the two registers serving as the taint cache.

## 7. EVALUATION

We evaluate the scalability of our instrumentation techniques on a collection of moderate-sized C and C++ applications, while using the SPEC benchmarks for performance evaluation. All our experiments were conducted on Ubuntu Linux 2.6.19. system with a 2.2 Ghz Intel Core 2 duo processor and 2GB of main memory.

### 7.1 Functionality

We used small-scale unit tests as well as utility applications on Linux to verify functional correctness of our techniques. In particular, we marked inputs to these utilities as tainted and verified that the outputs were tainted in the manner that we expect. We also checked that on the SPEC95 INT benchmarks, the outputs were tainted as expected when the inputs were (partially) tainted.

In addition, we verified that our technique can detect memory corruption attacks on two real-world HTTP servers, namely ATPhttpd 0.4b and GazTek ghttpd. ATPHttpd 0.4b and GazTek ghttpd 1.4 have expoitable buffer overflow vulnerabilities, referenced in CVE-2002-1816 and CVE-2002-1904 respectively. The exploit allows remote attackers to inject arbitrary code by providing excessively long HTTP GET payloads. Our dynamic tainting successfully detected that the network input taints the program counter value when the attack input is given to the server.

Finally, we verified that the technique is scalable and robust enough to handle moderately large applications written in C and C++, such as gimp-2.2 (an image editor), xmms (a music player), and a core library of Kpdf (a document viewer in KDE).

### 7.2 Analysis Time

Running times for the all analysis to complete are shown in Figure 4. The sizes of binary code for these applications varies between tens of KBs to a few MBs. This size includes binary code as well as static data, and hence does not always correlate well with the analysis time. For this reason, the table includes the number of instructions rather than the binary size.

In general, the analysis times were comparable to the build times for the respective applications, and are hence quite acceptable for a static instrumentation tool.

### 7.3 Runtime Performance

We used several CPU intensive programs from the SPEC95 INT benchmark suite for evaluating the effectiveness of our optimizations. (We did not evaluate performance overheads on the programs mentioned in the previous paragraph, as most of them are interactive GUI-based applications.) All these benchmarks were compiled with `gcc-4.1 -O2`.

Figure 5 shows the performance overheads for each of these applications, expressed as a ratio of their uninstrumented runtimes. The base numbers corresponds to the overheads before our optimizations are performed. Note that the base numbers do reflect some of the standard optimizations such as register liveness and dead code elimination, as well as some of the low-level optimizations outlined in [26]. As a result, the base figures reflect an average slowdown by a factor of about 5.8.

We then measured the overheads with the optimizations. First, the metadata caching optimization was applied. It leads to about 170% reduction in overhead, reducing it from a factor of about 4.8 to 3.1. This large decrease is explained by the following factors:

- The number of registers needed for metadata computation is decreased by the use of taint-stack, since there is no need to compute the address of tagmap locations corresponding to local variables. As a result, more registers can be spared for use as metadata caches, leading to significant reductions in load/stores of metadata.
- Since metadata-related address computations are avoided in most cases, the associated clobbering of CPU flags is

also avoided. As a result, expensive flag save/restore operations are significantly reduced. We found that less than 6% of the instructions retain the flag/save restore instrumentation after this optimization.

Tag-sharing optimization results in a further overhead reduction of about 120% on the average. As described before, this decrease results because of (a) elimination of metadata operations when the result can be statically computed, and (b) elimination of metadata moves associated with many of the data movement operations.

Finally, code specialization leads to significant additional reductions in overhead. The fraction of time spent in the two code versions is a function of the fraction of inputs that are tainted, which is in turn dependent on the application for which taint-tracking is being utilized. We therefore measured the performance of the fastpath and slowpath code separately. On the average, the fast path incurs about 90% overhead, while the slowpath incurs about twice this overhead. Previous research results [26] suggest that an overwhelming majority of execution uses fastpath, so the actual performance can be close to the fastpath figures.

Our performance results represent a significant improvement over that reported for LIFT [26]. In particular, our slowpath performance is about 3 times faster than their slowpath performance, while our fastpath performance is about twice as fast as theirs. Moreover, whereas their approach relies on the availability of additional CPU registers unused in original application code, our technique does not require such registers.

# 8. RELATED WORK

**Source-code based techniques for taint-tracking.**

Some of the efforts in taint-tracking [33, 17] have been based on a source-code transformation approach. These techniques are able to take advantage of high-level information available in source-code, and also "piggyback" on the optimization techniques implemented in compilers. In contrast, our techniques seek to achieve comparable performance while operating on binaries.

**Binary instrumentation for taint-tracking.**

Binary instrumentation for taint-tracking was first developed in [24]. While effective in attack detection, their approach slowed down programs significantly (often, by more than 20x), sparking interest in optimization techniques. TaintTrace [9] achieved significantly faster taint-tracking by using more efficient instrumentation based on DynamoRIO, combined with simple static analyses for eliminating redundant register saves and a shadow memory data structure (which is similar to our tagmap) that speeded up metadata access. LIFT [26] achieved significant additional performance benefits by using better static analysis and faster instrumentation techniques. As mentioned earlier, we adapted many of their low-level optimizations such as those involving the use of `lahf`/`sahf` instead of `pusha`/`popa`, but unlike them, we do not assume the availability of additional dedicated registers for instrumentation.

The primary difference between LIFT and our approach is that we develop sophisticated procedure-level static analyses that were not considered by them. These analyses have enabled us to improve over their performance by a factor of 2 to 3. Although our code specialization optimization is similar to theirs at the high level, there are significant differences in terms of the specifics, as described earlier.

**Static Analysis of Binaries.**

Previous techniques have applied abstract interpretation based analysis to recover some notion of higher level program variables on x86 binaries. Our stack analysis shares some of the goals of VSA [2], e.g., it reasons about integer values and pointer values simultaneously. But there are several important differences. Since our interest is in optimization, we are willing to sacrifice accuracy for analysis speed and scalability, as long as the analysis uncovers the most of the significant optimization opportunities, such as those involving stack-allocated memory. In contrast, VSA is focussed on much more accurate analysis, especially of global and heap memory; and to accurately identify variable-like entities in binary code [3]. From a technical point of view, our abstract domain uses symbolic values that allow us to reason about equality of variables at different program points, whereas their abstract domain is designed primarily to discover inequalities. Equalities are essential for reasoning about the side-effect of function calls on registers (such as ESP) and stack-based memory. Our approach uses a polymorphic function summaries which allows it to scale well to larger programs, whereas VSA analysis is context-sensitive.

**Binary instrumentation frameworks.**

Earliest static transformation tools such as EEL [18] were developed for binary transformation, but they did not consider the complexity of CISC architectures like x86. Recent robust tools such as Vulcan [30] offer static instrumentation capabilities, but assume the availability of much more high-level information such as variable boundaries.

Emulation based techniques such as Bochs [19] and QEMU [4] are good for whole system analysis, but for application specific information they offer a much coarser granularity (for instance, distinguishing OS and application data is challenging). Dynamic code transformation systems such as Pin [20], DynamoRIO [16], Valgrind [23], and Strata [28] have been widely used in security research, and offer good robustness for application transformation at the expense of higher overheads than ours, especially in applications that require extensive, fine-grained instrumentation.

BIRD [21] uses a combination of static and dynamic techniques to correctly disassemble large COTS binaries that are stripped of symbol information. Due to the modular nature of our analysis and optimization techniques, they can be easily combined with such hybrid disassembly techniques. As long as most of the code is statically disassembled, our technique can provide performance gains that are close to those reported here, while supporting stripped binaries.

Metadata access speedup techniques are proposed in [22]. Our techniques, which rely primarily on static analysis to improve metadata operations, are largely orthogonal to metadata maintenance schemes described by them.

# 9. CONCLUSION

In this paper, we presented techniques for optimizing fine-grained instrumentation of binaries. We developed a static analysis technique that enables sound optimizations on instrumentation code. Based on the results of this analysis, we presented several interesting optimization techniques for

improving the performance of metadata maintenance operations. Our evaluation results represent a substantial improvement in performance over that reported by previous works on binary instrumentation for taint-tracking. We believe that many of the techniques developed in the paper can be applied to other types of fine-grained instrumentations beyond taint-tracking.

## 10. REFERENCES


[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM CCS*, 2005.

[2] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. CC (LNCS 2985)*, 2004.

[3] G. Balakrishnan and T. Reps. Divine: Discovering variables in executables. In *VMCAI*, 2007.

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[5] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *14th USENIX Security Symposium*, Baltimore, MD, August 2005.

[6] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-based transparent and comprehensive control-flow error detection. In *CGO*, 2006.

[7] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. *LNCS*, 735, 1993.

[8] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2005.

[9] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *ISCC '06: Proceedings of the 11th IEEE Symposium on Computers and Communications*, pages 749–754, Washington, DC, USA, 2006. IEEE Computer Society.

[10] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *SOSP*, pages 133–147, New York, NY, USA, 2005. ACM Press.

[11] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, 1998.

[12] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Usenix Tech Conference*, 2007.

[13] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[14] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *SIGSOFT FSE*, 2006.

[15] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[16] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium.*, 2002.

[17] Lap Chung Lam and Tzi cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[18] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *PLDI*, 1995.

[19] Kevin P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux J.*, page 7.

[20] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[21] Susanta Nanda, Wei Li, Lap chung Lam, and Tzi cker Chiueh. BIRD: Binary interpretation using runtime disassembly. In *IEEE/ACM Conference on Code Generation and Optimization (CGO)*, March 2006.

[22] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *ACM/USENIX Conference on Virtual Execution Environments*, 2007.

[23] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[24] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium*, 2005.

[25] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *RAID*, 2005.

[26] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting general security attacks. In *IEEE/ACM MICRO*, 2006.

[27] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited, 2002.

[28] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO generation and optimization*, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

[29] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference*, 2005.

[30] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, 2001.

[31] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.

[32] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of NDSS*, San Diego, CA, February 2007.

[33] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, August 2006.

[34] L. Xun. A Linux executable editing library (LEEL), 1999.

[35] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS*, 2007.

[36] Yves Younan, Davide Pozza, Wouter Joosen, and Frank Piessens. Extended protection against stack smashing attacks without performance loss. In *Proceedings of ACSAC*, 2006.