

MODERN MEMORY DEFENSES

GRAD SEC
SEP 14 2017



TODAY'S PAPERS

Verizon 6:52 AM 100%

cs.umd.edu

Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software

Jarvis Newsome
newsome@cs.cmu.edu
Carnegie Mellon University

Drew Song
dawnsong@cmu.edu
Carnegie Mellon University

Abstract

Software vulnerabilities have had a devastating effect on the Internet. Worms such as CodeRed and Slammer can compromise hundreds of thousands of hosts within hours or even minutes, and cause millions of dollars of damage [26, 43]. To successfully combat these fast automatic Internet attacks, we need fast automatic attack detection and filtering mechanisms.

In this paper we propose dynamic taint analysis for automatic detection of executable attacks, which include most types of exploits. This approach does not need source code or special compilation for the monitored program, and hence works on commodity software. To demonstrate this idea, we have implemented *TaintCheck*, a mechanism that can perform dynamic taint analysis by performing binary rewriting at runtime. We show that *TaintCheck* reliably detects most types of exploits. We prove that *TaintCheck* produces no false positives for any of the many different programs that we tested. Further, we describe how *TaintCheck* could improve automatic signature generation in several ways.

1. Introduction

Software vulnerabilities such as buffer overruns and format string vulnerabilities have had a devastating effect on the Internet. Worms such as CodeRed and Slammer exploit software vulnerabilities and can compromise tens of thousands of hosts within hours or even minutes, and cause millions of dollars of damage [26, 43]. To successfully combat fast Internet worm attacks, we need automatic detection and defense mechanisms. First, we need automatic detection mechanisms that can detect new attacks for previously unknown vulnerabilities. A detection mechanism should be easy to deploy, result in few false positives and few false negatives, and detect attacks only before a significant fraction of vulnerable systems are compromised. Second, once a new exploit attack is

detected, we must quickly develop fixes (aka. attack signatures) that can be used to filter out attack packets efficiently, and hence protect vulnerable systems from compromise until the vulnerability can be patched. Because a new worm can spread quickly, signature generation must be automatic—no manual intervention can respond quickly enough to prevent a large number of vulnerable hosts from being infected by a new fast-spreading worm.

We need fine-grained attack detectors for commodity software. Many approaches have been proposed to detect new attacks. These approaches roughly fall into two categories: coarse-grained detectors, that detect anomalous behavior, such as scanning or unusual activity at a certain point, and fine-grained detectors, that detect attacks on a program's vulnerabilities. Coarse-grained detectors may result in frequent false positives, and do not provide detailed information about the vulnerability and how it is exploited. Thus, it is desirable to develop fine-grained detectors that produce fewer false positives, and provide detailed information about the vulnerability and exploit.

Several approaches for fine-grained detection have been proposed that detect when a program is exploited. Most of these previous mechanisms require source code or special recompilation of the program, such as StackGuard [15], PointGuard [14], full-bound check [26, 31], Libsafe-Plus [5], FortifySource [10], and CCheck [20]. Some of them also require recompiling the libraries [20, 38] or modifying the original source code, or are not compatible with some programs [25, 14]. These constraints hinder the deployment and applicability of these methods, especially for commodity software, because source code or specially-compiled binaries are often unavailable, and the additional work required (such as recompiling the libraries and modifying the original source code) makes it inconvenient to apply these methods to a broad range of applications. Note that most of the large-scale worms attack to Internet attacks on commodity software.

Thus, it is important to design fine-grained detectors that work on commodity software, i.e., work on arbitrary

code without requiring source code or specially-compiled binaries. This goal is difficult to achieve because important information, such as type information, is not generally available in binaries. As a result, we

write signatures for attack filtering. We have developed an automatic tool, *TaintCheck*, to demonstrate our dynamic taint analysis approach. *TaintCheck* offers several unique benefits.

Verizon 6:52 AM 100%

cs.umd.edu

Control-Flow Integrity

Principles, Implementations, and Applications

Martin Abadi
Computer Science Dept.
University of California
Santa Cruz

Mhai Budiu
Microsoft Research
Silicon Valley

Ulfar Erlingsson
Microsoft Research
Silicon Valley

Jay Ligatti
Dept. of Computer Science
Simon Fraser University

ABSTRACT

Current software attacks often build on exploits that subvert machine-code execution. The enforcement of a basic safety property, Control-Flow Integrity (CFI), can prevent such attacks from arbitrarily subverting program behavior. CFI enforcement is simple and its parameters can be automatic, even with respect to powerful adversaries. Moreover, CFI enforcement is practical: it is compatible with existing software and can be done efficiently using software rewriting in commodity systems. Finally, CFI provides a useful foundation for enforcing further security policies, as we demonstrate with efficient software implementations of a protocol-shield call stack and of access control for memory regions.

Categories and Subject Descriptors

D.1.4 [Programming Languages]: Descriptors, D.2.4 [Software Engineering]: Software/Program Verification, D.1.6 [Operating Systems]: Security and Protection

General Terms

Security, Languages, Verification

Keywords

Binary Rewriting, Control-Flow Graph, Mutual-Reference, Mutation, Vulnerabilities

hard effects of these attacks make them one of the most pressing challenges in computer security.

In recent years, many signature-vulnerability mitigations have been proposed for defending against these attacks: these include stack canaries [11], runtime elimination of buffer overflows [46], randomization and artificial homogeneity [41, 67], and clearing of suspect data [5]. Some of these mitigations are widely used, while others may be impractical, for example because they rely on hardware modifications or impose a high performance penalty. In any case, their security benefits are open to debate: mitigations are usually of limited scope, and attacks have been found to circumvent each deployed mitigation mechanism [42, 49, 41].

The intuition of these mechanisms stem, in part, from the lack of a realistic attack model and the reliance on informal reasoning and hidden assumptions. In order to be trustworthy, mitigation techniques should—given the legitimacy of possible attacks and the wealth of current and undiscovered software vulnerabilities—be simple to comprehend and to enforce, yet provide strong guarantees against powerful adversaries. On the other hand, in order to be deployable in practice, mitigation techniques should be applicable to existing code (preferably even to legacy binaries) and incur low overhead.

This paper describes and studies one mitigation technique, the enforcement of Control-Flow Integrity (CFI) that aims to meet these standards for trustworthiness and deployability. The paper introduces CFI enforcement, presents an implementation for Windows on the x86 architecture, gives results from experiments, and suggests applications.

The CFI security policy dictates that software execution must follow a path of a Control-Flow Graph (CFG) determined ahead of time. The CFG in question can be defined by analysis—source-code analysis, binary analysis, or execution profiling. For our experiments, we focus on CFGs that are derived by a static binary analysis. CFGs can also be defined by explicit security policies, for example written as security automata [17].

A security policy is of limited value without an attack model. In our design, CFI enforcement provides protection even against powerful adversaries that have full control over the entire data memory of the executing program. This model of adversarial may seem rather pessimistic. On the other hand, it is a number of virtuals. First, it is clear, and reasonable, to demand definition and analysis. It also allows for the real possibility that buffer overflows or other vulnerabilities (e.g., [26]) would lead to arbitrary changes in data memory. Finally, it applies even when an attacker is in possession of a module or thread within the same address space as the program being protected.

Whereas CFI enforcement can potentially be done in several ways, we try as a combination of lightweight static verification

and machine-code rewriting that instruments software with runtime checks. The runtime checks dynamically ensure that control flow remains within a given CFG. As we demonstrate, machine-code rewriting results in a practical implementation of CFI enforcement. This implementation applies to existing source-level programs on commodity systems, and yields efficient code even in irregular architectures such as x86-64, ARMv7, and MIPS64.

2. RELATED WORK

Our work on CFI is related to many techniques that, either directly or indirectly, constrain control flow. For the purposes of the present section, we divide these techniques according to whether they aim to achieve security on individual systems.

CONTROL FLOW INTEGRITY

Fundamentally, code injection attacks altered the target program's control flow

Recall: Confidentiality, Integrity, Availability

Most integrity defenses seek to detect

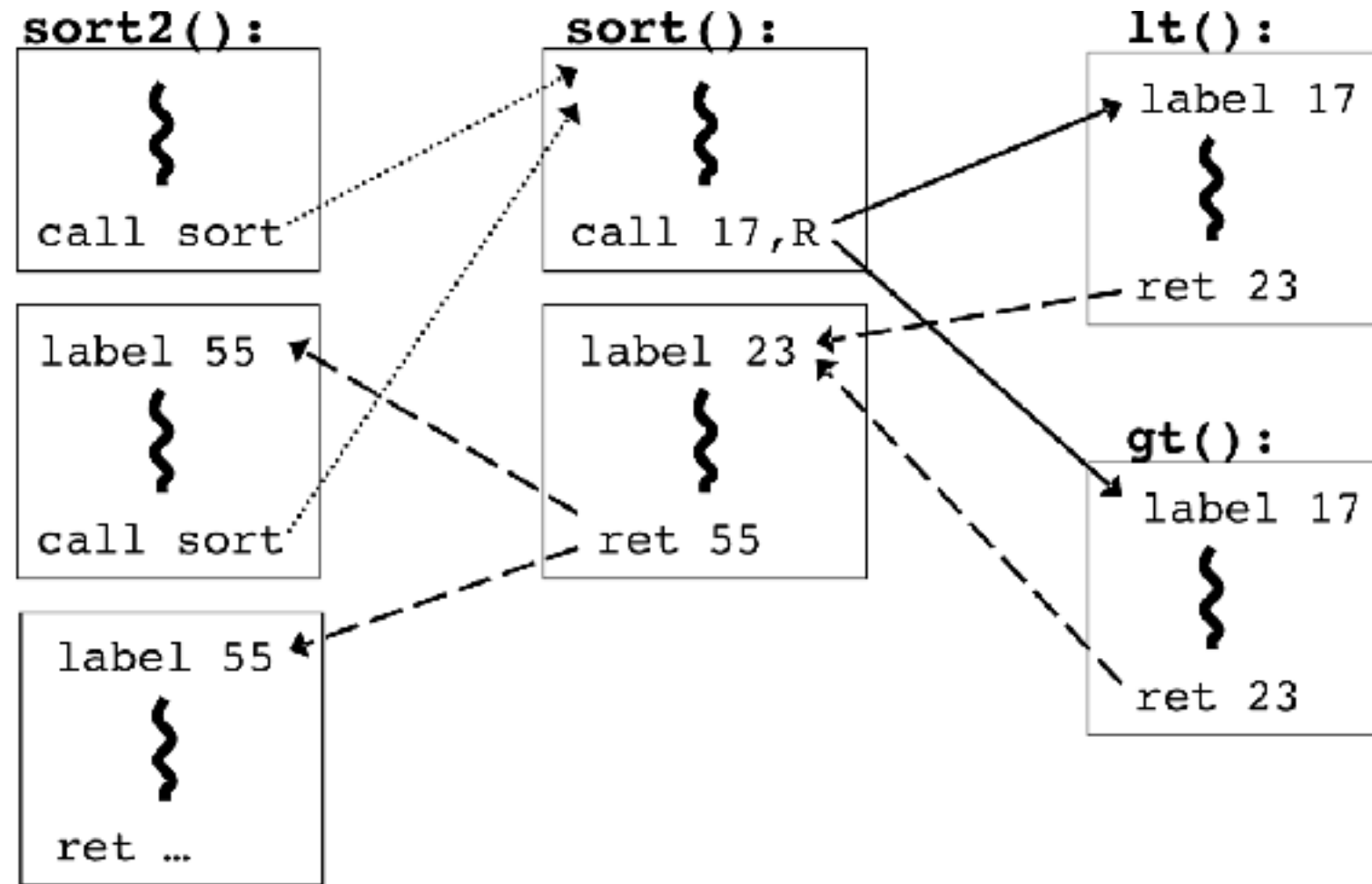
Typically they are unable to outright prevent

CONTROL FLOW GRAPH

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



Code injection, return to libc, ROP.. all of them alter where one of the "ret"s points

REFERENCE MONITORS

Code or system responsible for checking whether data/execution matches some policy

File permissions, password checker, airline employees checking tickets...

Mediates between user and sensitive resource

CFI is an *inline* reference monitor

ENSURE COMPLETE MEDIATION



SOFTWARE FAULT ISOLATION (SFI)

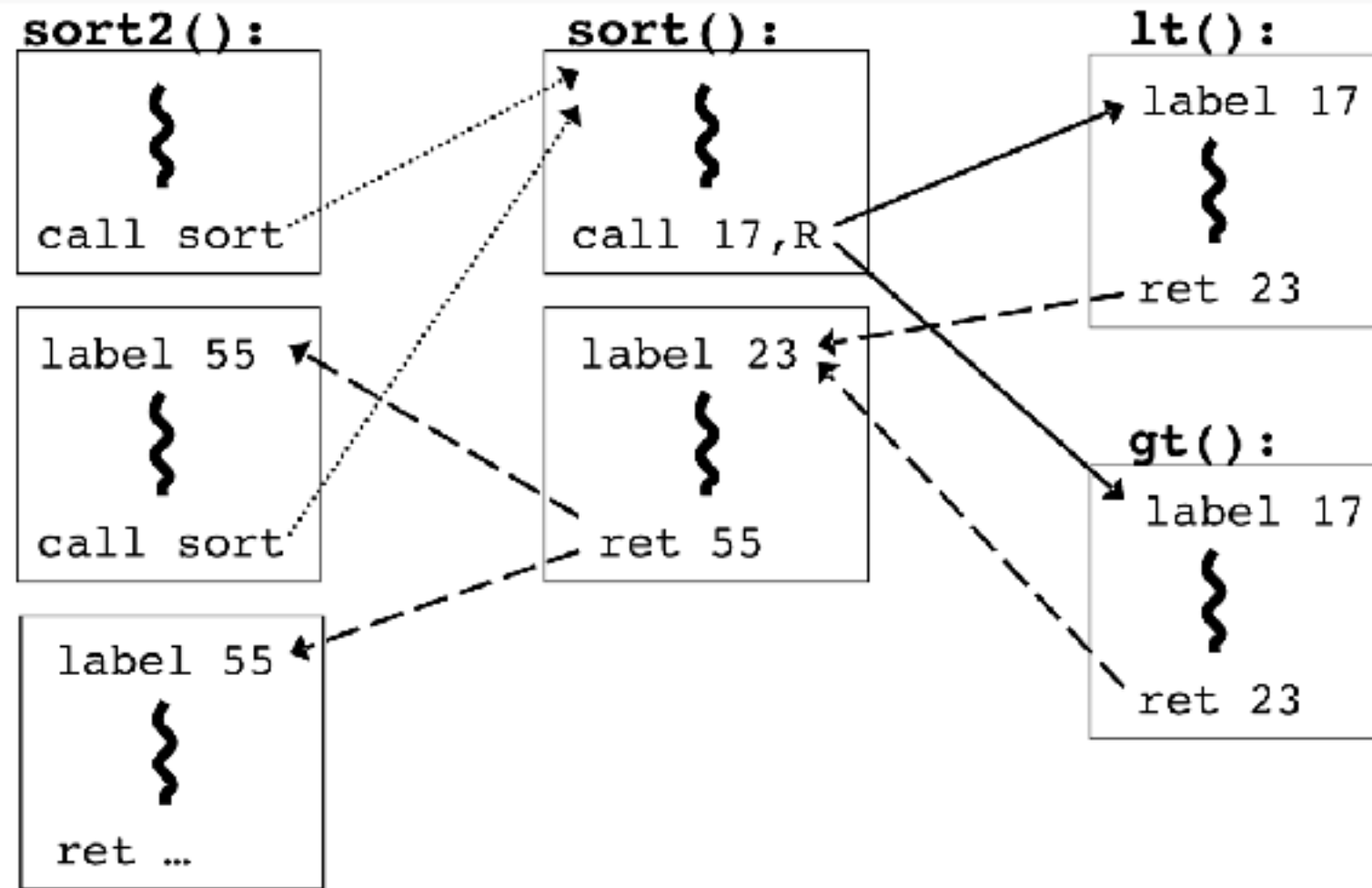
Insert code at each machine code instruction to ensure that the target memory region lies within some bounds

```
    ...  
    mov    ecx, 0h           ; int i = 0  
    mov    esi, [esp+8]     ; a[] base ptr  
    and    esi, 20FFFFFFh   ; SFI masking  
LOOP: add    eax, [esi+ecx*4] ; sum += a[i]  
    inc    ecx             ; ++i  
    cmp    ecx, edx        ; i < len  
    jl    LOOP
```

**Keep only the LSBs (zero with 'and' then
add the target memory region's MSBs**

INTEGRITY WITH LABELS

```
bool lt(int x, int y) {  
    return x < y;  
}  
  
bool gt(int x, int y) {  
    return x > y;  
}  
  
sort2(int a[], int b[], int len)  
{  
    sort( a, len, lt );  
    sort( b, len, gt );  
}
```



Note that we start in the trusted code.

The goal is to make sure we never ret somewhere we shouldn't

INLINING CFI

Source		Destination	
Opcode bytes	Instructions	Opcode bytes	Instructions
FF E1	jmp ecx ; computed jump	8B 44 24 04	mov eax, [esp+4] ; dst
		...	
can be instrumented as (a):			
81 39 78 56 34 12	cmp [ecx], 12345678h ; comp ID & dst	78 56 34 12	; data 12345678h ; ID
75 13	jne error_label ; if != fail	8B 44 24 04	mov eax, [esp+4] ; dst
8D 49 04	lea ecx, [ecx+4] ; skip ID at dst	...	
FF E1	jmp ecx ; jump to dst		
or, alternatively, instrumented as (b):			
B8 77 56 34 12	mov eax, 12345677h ; load ID-1	3E 0F 18 05	prefetchnta ; label
40	inc eax ; add 1 for ID	78 56 34 12	[12345678h] ; ID
39 41 04	cmp [ecx+4], eax ; compare w/dst	8B 44 24 04	mov eax, [esp+4] ; dst
75 13	jne error_label ; if != fail	...	
FF E1	jmp ecx ; jump to label		

Figure 2: Example CFI instrumentations of a source x86 instruction and one of its destinations.

Will only jump to a part of the code with the label 0x12345678

SECURITY GUARANTEES

Attack model: arbitrary control over the data portion of memory

UNQ: No label appears elsewhere in code

NWC: Code segment is not writable

NXD: Data segment is not executable

SOFTWARE FAULT ISOLATION (SFI)

Insert code at each machine code instruction to ensure that the target memory region lies within some bounds

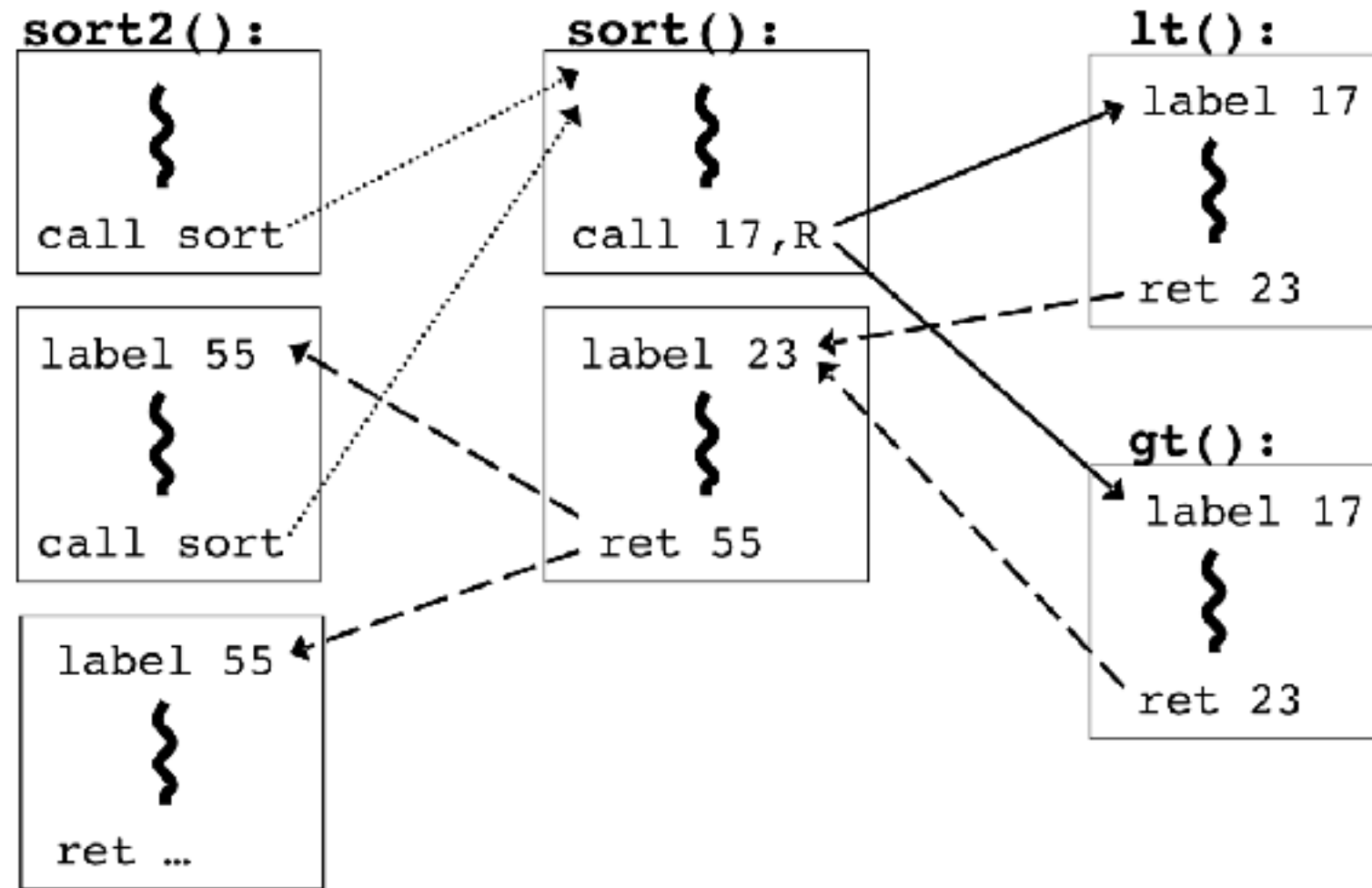
```
    . . .
    mov    ecx, 0h                ; int i = 0
    mov    esi, [esp+8]          ; a[] base ptr
    and    esi, 20FFFFFFh        ; SFI masking
LOOP: add    eax, [esi+ecx*4]     ; sum += a[i]
    inc    ecx                   ; ++i
    cmp    ecx, edx              ; i < len
    jl     LOOP
```

Normally you want the ‘and’ in the loop,

But CFI ensures no jumps into the loop

LABELS ARE NOT UNIQUE

```
bool lt(int x, int y) {  
    return x < y;  
}  
  
bool gt(int x, int y) {  
    return x > y;  
}  
  
sort2(int a[], int b[], int len)  
{  
    sort( a, len, lt );  
    sort( b, len, gt );  
}
```



Attacker could potentially cause `sort()` to return to either of the memory locations labelled 55

LABELS ARE NOT UNIQUE

Code duplication

Shadow stack

SHADOW CALL STACKS

One possibility: SFI to maintain a region of memory (e.g., 0x1*) specifically for the shadow call stack

Hardware support: x86 offers memory segments

```
call  eax                ; call func ptr                ret                ; return
```

with a CFI-based implementation of a protected shadow call stack using hardware segments, can become:

```
add  gs:[0h], 4h        ; inc stack by 4                mov  ecx, gs:[0h]      ; get top offset
mov  ecx, gs:[0h]       ; get top offset                mov  ecx, gs:[ecx]    ; pop return dst
mov  gs:[ecx], LRET     ; push ret dst                    sub  gs:[0h], 4h      ; dec stack by 4
cmp  [eax+4], ID        ; comp fptr w/ID                 add  esp, 4h          ; skip extra ret
jne  error_label       ; if != fail                       jmp  ecx              ; jump return dst
call eax                ; call func ptr
```

LRET: ...

%gs always points to shadow stack segment

Protected by CFI + static analysis of code

SECURITY GUARANTEES

Attack model: arbitrary control over the data portion of memory

UNQ: No label appears elsewhere in code

NWC: Code segment is not writable

NXD: Data segment is not executable

Let S_0 be a state with code memory M_c such that $I(M_c)$ and $pc = 0$, and let S_1, \dots, S_n be states such that $S_0 \rightarrow S_1 \rightarrow \dots \rightarrow S_n$. Then, for all $i \in 0..(n - 1)$, either $S_i \rightarrow_a S_{i+1}$ or the pc at S_{i+1} is one of the allowed successors for the pc at S_i according to the given CFG.

EVALUATION

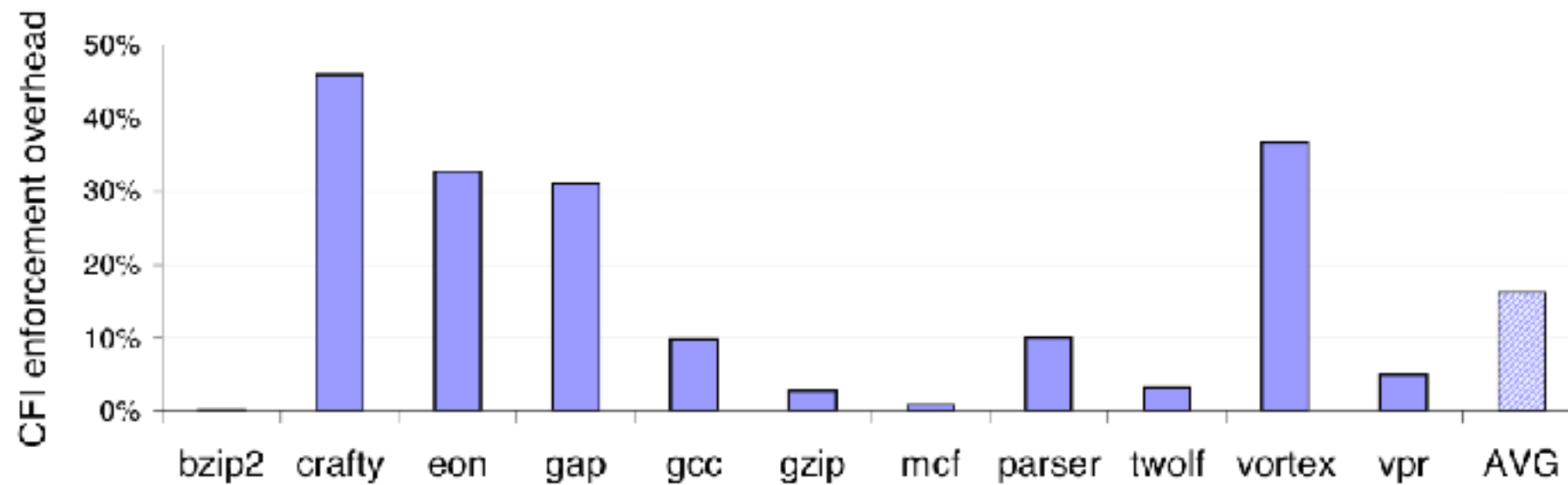


Figure 4: Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

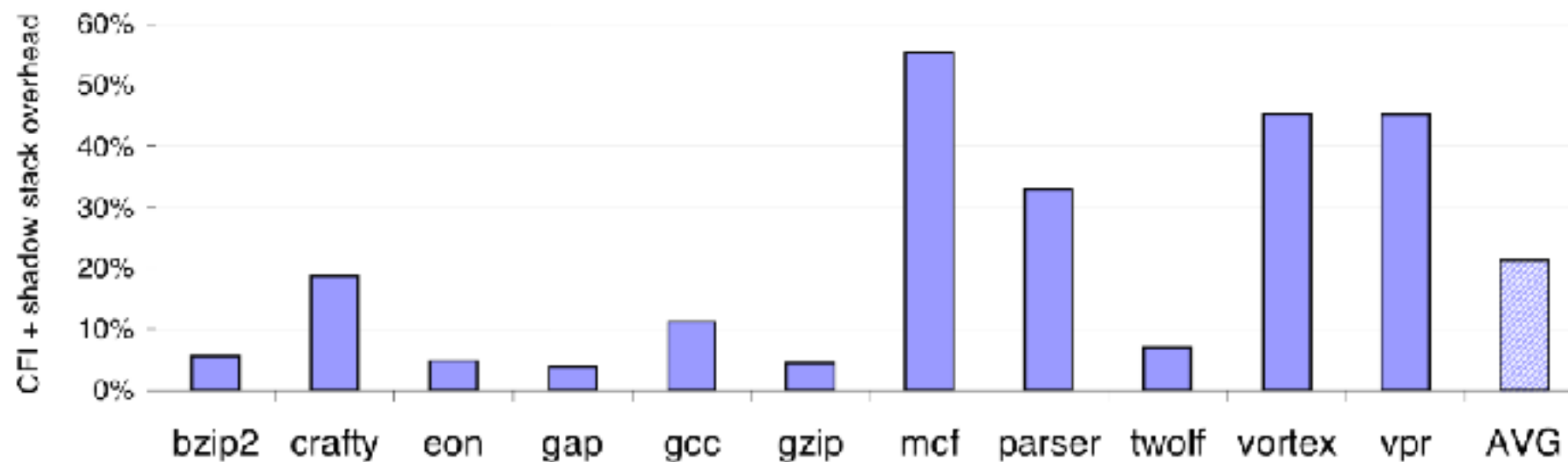


Figure 8: Enforcement overhead for CFI with a protected shadow call stack on SPEC2000 benchmarks.

Shadow stack reduces some unnecessary ID checks during returns

CFI: SHORTCOMINGS

CFI: SHORTCOMINGS

No dynamically generated code (functional programming?)

Requires recompiling the code

TODAY'S PAPERS

Verizon 6:52 AM 100%

cs.umd.edu

Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software

Jarvis Newsome
newsome@cs.cmu.edu
Carnegie Mellon University

Drew Song
dawnsong@cmu.edu
Carnegie Mellon University

Abstract

Software vulnerabilities have had a devastating effect on the Internet. Worms such as CodeRed and Slammer can compromise hundreds of thousands of hosts within hours or even minutes, and cause millions of dollars of damage [26, 43]. To successfully combat these fast automatic Internet attacks, we need fast automatic attack detection and filtering mechanisms.

In this paper we propose dynamic taint analysis for automatic detection of executable attacks, which include most types of exploits. This approach does not need source code or special compilation for the monitored program, and hence works on commodity software. To demonstrate this idea, we have implemented *TaintCheck*, a mechanism that can perform dynamic taint analysis by performing binary rewriting at runtime. We show that *TaintCheck* reliably detects most types of exploits. We prove that *TaintCheck* produces no false positives for any of the many different programs that we tested. Further, we describe how *TaintCheck* could improve automatic signature generation in several ways.

1. Introduction

Software vulnerabilities such as buffer overruns and format string vulnerabilities have had a devastating effect on the Internet. Worms such as CodeRed and Slammer exploit software vulnerabilities and can compromise tens of thousands of hosts within hours or even minutes, and cause millions of dollars of damage [26, 43]. To successfully combat fast Internet worm attacks, we need automatic detection and defense mechanisms. First, we need automatic detection mechanisms that can detect new attacks for previously unknown vulnerabilities. A detection mechanism should be easy to deploy, result in few false positives and few false negatives, and detect attacks only before a significant fraction of vulnerable systems are compromised. Second, once a new exploit attack is

detected, we must quickly develop fixes (aka. attack signatures) that can be used to filter out attack packets efficiently, and hence protect vulnerable systems from compromise until the vulnerability can be patched. Because a new worm can spread quickly, signature generation must be automatic—no manual intervention can respond quickly enough to prevent a large number of vulnerable hosts from being infected by a new fast-spreading worm.

We need fine-grained attack detectors for commodity software. Many approaches have been proposed to detect new attacks. These approaches roughly fall into two categories: coarse-grained detectors, that detect anomalous behavior, such as scanning or unusual activity at a certain point, and fine-grained detectors, that detect attacks on a program's vulnerabilities. Coarse-grained detectors may result in frequent false positives, and do not provide detailed information about the vulnerability and how it is exploited. Thus, it is desirable to develop fine-grained detectors that produce fewer false positives, and provide detailed information about the vulnerability and exploit.

Several approaches for fine-grained detection have been proposed that detect when a program is exploited. Most of these previous mechanisms require source code or special recompilation of the program, such as StackGuard [15], PointGuard [14], full-bound check [26, 31], Libsafe-Plus [5], FortifySource [10], and CCheck [20]. Some of them also require recompiling the libraries [20, 38] or modifying the original source code, or are not compatible with some programs [25, 14]. These constraints hinder the deployment and applicability of these methods, especially for commodity software, because source code is especially recompiled binaries are often unavailable, and the additional work required (such as recompiling the libraries and modifying the original source code) makes it inconvenient to apply these methods to a broad range of applications. Note that most of the large-scale worms attack to Internet attacks on commodity software.

Thus, it is important to design fine-grained detectors that work on commodity software, i.e., work on arbitrary

code without requiring source code or specially recompiled binaries. This goal is difficult to achieve because important information, such as type information, is not generally available in binaries. As a result, we

write signatures for attack filtering. We have developed an automatic tool, *TaintCheck*, to demonstrate our dynamic taint analysis approach. *TaintCheck* offers several unique benefits.

Verizon 6:52 AM 100%

cs.umd.edu

Control-Flow Integrity

Principles, Implementations, and Applications

Martin Abadi
Computer Science Dept.
University of California
Santa Cruz

Mhai Budiu
Microsoft Research
Silicon Valley

Ulfar Erlingsson
Microsoft Research
Silicon Valley

Jay Ligatti
Dept. of Computer Science
Simon Fraser University

ABSTRACT

Current software attacks often build on exploits that subvert machine-code execution. The enforcement of a basic safety property Control-Flow Integrity (CFI), can prevent such attacks from arbitrarily subverting program behavior. CFI enforcement is simple and its parameters can be automatic, even with respect to powerful adversaries. Moreover, CFI enforcement is practical: it is compatible with existing software and can be done efficiently using software rewriting in commodity systems. Finally, CFI provides a useful foundation for enforcing further security policies, as we demonstrate with efficient software implementations of a protocol-based call stack and of access control for memory regions.

Categories and Subject Descriptors

D.1.4 [Programming Languages]: Descriptors, D.2.4 [Software Engineering]: Software/Program Verification, D.1.6 [Operating Systems]: Security and Protection

General Terms

Security, Languages, Verification

Keywords

Binary Rewriting, Control-Flow Graph, Mutual-Reference, Mutation, Vulnerabilities

hard effects of these attacks make them one of the most pressing challenges in computer security.

In recent years, many automatic vulnerability mitigation have been proposed for defending against these attacks: these include stack canaries [11], runtime elimination of buffer overflows [46], randomization and artificial homogeneity [41, 67], and clearing of suspect data [5]. Some of these mitigations are widely used, while others may be impractical, for example because they rely on hardware modifications or impose a high performance penalty. In any case, their security benefits are open to debate: mitigations are usually of limited scope, and attacks have been found to circumvent each deployed mitigation mechanism [42, 49, 41].

The intuition of these mechanisms stem, in part, from the lack of a realistic attack model and the reliance on informal reasoning and hidden assumptions. In order to be trustworthy, mitigation techniques should—given the legitimacy of possible attacks and the wealth of current and undiscovered software vulnerabilities—be simple to comprehend and to enforce, yet provide strong guarantees against powerful adversaries. On the other hand, in order to be deployable in practice, mitigation techniques should be applicable to existing code (preferably even to legacy binaries) and incur low overhead.

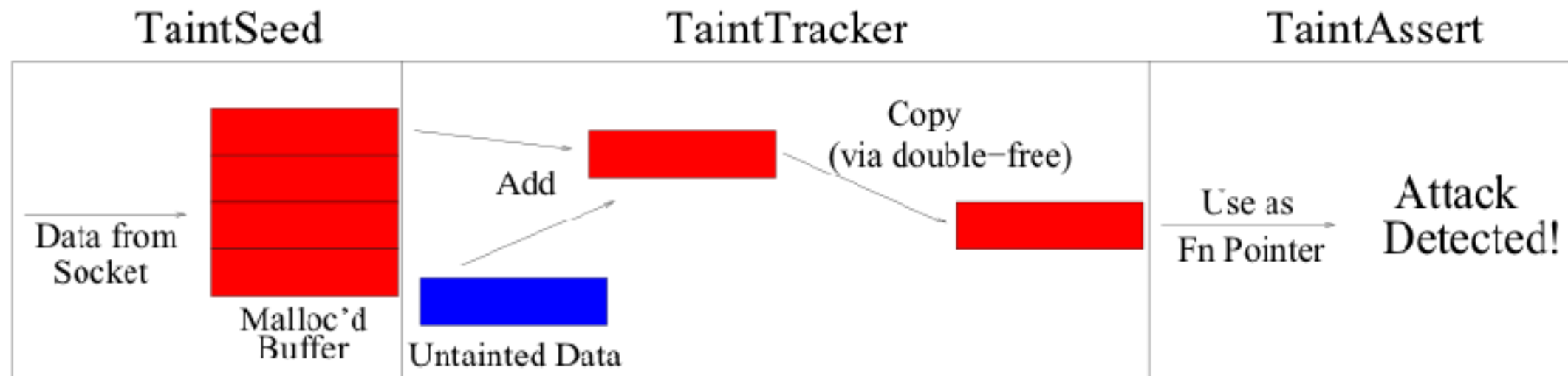
This paper describes and studies one mitigation technique, the enforcement of Control-Flow Integrity (CFI) that aims to meet these standards for trustworthiness and deployability. The paper introduces CFI enforcement, presents an implementation for Windows on the x86 architecture, gives results from experiments, and suggests applications.

The CFI security policy dictates that software execution must follow a path of a Control-Flow Graph (CFG) determined ahead of time. The CFG in question can be defined by analysis—source code analysis, binary analysis, or execution profiling. For our experiments, we focus on CFGs that are derived by a static binary analysis. CFGs can also be defined by explicit security policies, for example written as security automata [17].

A security policy is of limited value, unless it is attack-proof. In our design, CFI enforcement provides protection even against powerful adversaries that have full control over the entire data memory of the executing program. This model of adversarial may seem rather pessimistic. On the other hand, it is a number of virtuals. First, it is clear, and reasonable, to demand definitions and analysis. It also allows for the real possibility that buffer overflows or other vulnerabilities (e.g., [26]) would lead to arbitrary changes in data memory. Finally, it applies even when an attacker is in possession of a module or thread within the same address space as the program being protected.

Whereas CFI enforcement can potentially be done in several ways, we try as a combination of lightweight static verification

TAINT TRACKING: HIGH LEVEL IDEA



Potentially malicious input "taints" memory

Track what gets tainted

Enforce that some operations only work on untainted data

TAINT TRACKING: CHALLENGES

How do we track memory accesses?

How do we keep track of what's tainted?

How do we protect the taint info?

How do we “propagate” taint?

TAINT PROPAGATION (TAINTDROID)

Table 1: DEX Taint Propagation Logic. Register variables and class fields are referenced by v_X and f_X , respectively. R and E are the return and exception variables maintained within the interpreter. A , B , and C are byte-code constants.

Op Format	Op Semantics	Taint Propagation	Description
<i>const-op</i> $v_A C$	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
<i>move-op</i> $v_A v_B$	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
<i>move-op-R</i> v_A	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
<i>return-op</i> v_A	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint (\emptyset if void)
<i>move-op-E</i> v_A	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
<i>throw-op</i> v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
<i>unary-op</i> $v_A v_B$	$v_A \leftarrow \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
<i>binary-op</i> $v_A v_B v_C$	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	Set v_A taint to v_B taint \cup v_C taint
<i>binary-op</i> $v_A v_B$	$v_A \leftarrow v_A \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$	Update v_A taint with v_B taint
<i>binary-op</i> $v_A v_B C$	$v_A \leftarrow v_B \otimes C$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
<i>aput-op</i> $v_A v_B v_C$	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[\cdot]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A)$	Update array v_B taint with v_A taint
<i>uget-op</i> $v_A v_B v_C$	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_C)$	Set v_A taint to array and index taint
<i>sput-op</i> $v_A f_B$	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A taint
<i>sget-op</i> $v_A f_B$	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B taint
<i>iput-op</i> $v_A v_B f_C$	$v_B(f_C) \leftarrow v_A$	$\tau(v_B(f_C)) \leftarrow \tau(v_A)$	Set field f_C taint to v_A taint
<i>iget-op</i> $v_A v_B f_C$	$v_A \leftarrow v_B(f_C)$	$\tau(v_A) \leftarrow \tau(v_B(f_C)) \cup \tau(v_B)$	Set v_A taint to field f_C and object reference taint

Define what propagation rules for all operations

TAINT TRACKING

Instrument every (relevant) operation

Mechanism: Valgrind

Translates x86 into its own instruction set

Passes these to TaintCheck

TaintCheck passes back modified instructions

Add code to update taint info

TAINT STORING: RETURN OF THE SHADOW

1 byte memory -> 4 byte pointer -> taint data structure

Each byte of memory, including the registers, stack, heap, *etc.*, has a four-byte shadow memory that stores a pointer to a Taint data structure if that location is tainted, or a NULL pointer if it is not. We use a page-table-like structure to ensure that the shadow memory uses very little memory in practice. TaintSeed examines the arguments and results of each system call, and determines whether any memory written by the system call should be marked as tainted or untainted according to the TaintSeed policy. When the memory is tainted, TaintSeed allocates a Taint data structure that records the system call number, a snapshot of the current stack, and a copy of the data

POLICY CHECKING

Must specify what operations aren't permitted
on tainted data

EVALUATION

Has the possibility for false positives, false negatives

Program	Overwrite Method	Overwrite Target	Detected
ATPhttpd	buffer overflow	return address	✓
synthetic	buffer overflow	function pointer	✓
synthetic	buffer overflow	format string	✓
synthetic	format string	none (info leak)	✓
cfingerd	syslog format string	GOT entry	✓
wu-ftp	vsnprintf format string	return address	✓

Table 1. Evaluation of TaintCheck's ability to detect exploits

EVALUATION

Has the possibility to adversely affect performance

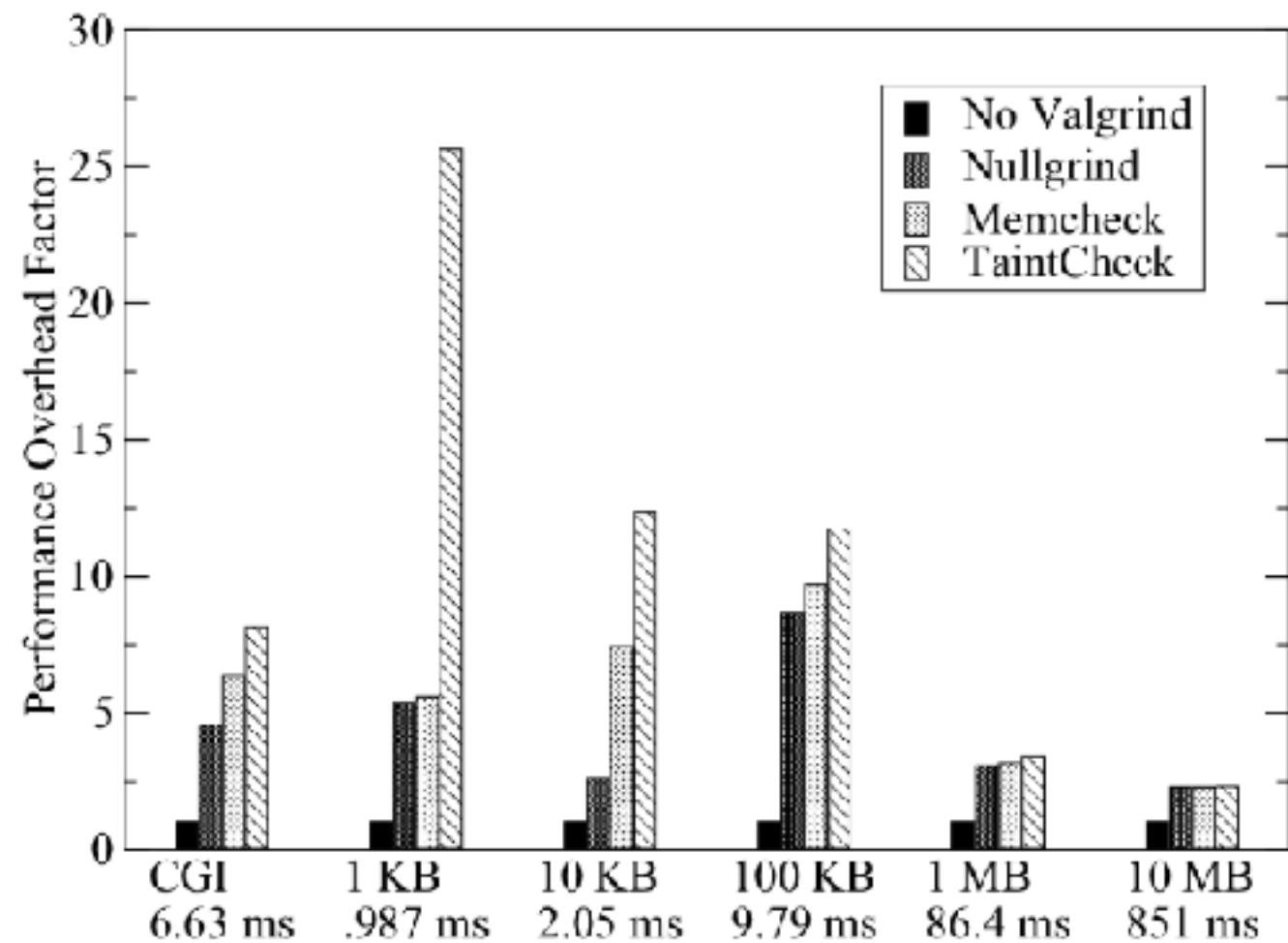


Figure 3. Performance overhead for Apache. Y-axis is the performance overhead factor: execution time divided by native execution time. Native execution times are listed below each experiment.

EVALUATION

Has the possibility to be overtrained to known vulnerabilities

TAINTDROID

Table 2: Applications grouped by the requested permissions (L: location, C: camera, A: audio, P: phone state). Android Market categories are indicated in parenthesis, showing the diversity of the studied applications.

Applications*	#	Permissions†			
		L	C	A	P
The Weather Channel (News & Weather); Cestos, Solitaire (Game); Movies (Entertainment); Babble (Social); Manga Browser (Comics)	6	x			
Bump, Wertago (Social); Antivirus (Communication); ABC — Animals, Traffic Jam, Hearts, Blackjack, (Games); Horoscope (Lifestyle); Yellow Pages (Reference); 3001 Wisdom Quotes Lite, Dastelefonbuch, Astrid (Productivity), BBC News Live Stream (News & Weather); Ringtones (Entertainment)	14	x			x
Layar (Lifestyle); Knocking (Social); Coupons (Shopping); Trapster (Travel); Spongebob Slide (Game); ProBasketBall (Sports)	6	x	x		x
MySpace (Social); Barcode Scanner, ixMAT (Shopping)	3		x		
Evernote (Productivity)	1	x	x	x	

* Listed names correspond to the name displayed on the phone and not necessarily the name listed in the Android Market.

† All listed applications also require access to the Internet.

TAINTDROID

Table 3: Potential privacy violations by 20 of the studied applications. Note that three applications had multiple violations, one of which had a violation in all three categories.

Observed Behavior (# of apps)	Details
Phone Information to Content Servers (2)	2 apps sent out the phone number, IMSI, and ICC-ID along with the geo-coordinates to the app's content server.
Device ID to Content Servers (7)*	2 Social, 1 Shopping, 1 Reference and three other apps transmitted the IMEI number to the app's content server.
Location to Advertisement Servers (15)	5 apps sent geo-coordinates to ad.qwapi.com, 5 apps to admob.com, 2 apps to ads.mobclix.com (1 sent location both to admob.com and ads.mobclix.com) and 4 apps sent location [†] to data.flurry.com.

* TaintDroid flagged nine applications in this category, but only seven transmitted the raw IMEI without mentioning such practice in the EULA.

[†]To the best of our knowledge, the binary messages contained tainted location data (see the discussion below).