Static Binary Analysis And Transformation For Sandboxing Untrusted Plugins

A Thesis Presented
by

Prateek Saxena

to

The Graduate School
in partial fulfillment of the
Requirements
for the degree of

Master of Science
in
Computer Science

Stony Brook University
August 2007

Stony Brook University

The Graduate School

Prateek Saxena

We, the thesis committee for the above candidate for the

Master of Science degree,

hereby recommend acceptance of this thesis.

Dr. R. C. Sekar, Thesis Advisor,
Computer Science Department

Dr. Scott Stoller, Chairman of Thesis Committee,
Computer Science Department

Dr. Rob Johnson, Committee Member,
Computer Science Department

This thesis is accepted by the Graduate School.

Dean of the Graduate School

Abstract of the Thesis

Static Binary Analysis And Transformation For Sandboxing Untrusted Plugins

by

Prateek Saxena

Master of Science

in

Computer Science

Stony Brook University

2007

Computers today have become a integral part of daily activities for users who rely on them as means of communication, financial management, entertainment, and business. Moreover, users today are depending increasingly on off-the-shelf software from untrusted sources like the Internet for everyday life. This factor has prompted research on mitigating the threat of untrusted programs running on the end user's personal computers. Relatively lesser focus has been laid on the threat of software extensions such as plug-ins and modules for trusted host system such as web browsers and email clients. With the alarming increase in malware that continues to defeat state-of-the-art defenses and evades detection, this work aims to be part of a general body of "proactive" defense technique that gives strong guarantees against future attacks, as opposed to a defense strategy that is developed in response to known vulnerabilities and their exploits.

In this context, this thesis has three objectives. First, it analyzes the threat model imposed by shared-address space extensions and modules that plug into larger host systems. Second, it surveys the limitations of existing mechanisms to deal with this threat, and proposes a practical approach to confinement of untrusted extensions. Finally, it presents a robust static binary rewriting and analysis framework that has more general applicability as tool for analysis and instrumentation for security applications where no source code is available.

Table of Contents

Chapter 1

Introduction

In recent years, there has been an explosive increase in malware, which often hides in software from untrusted sources on the Internet. This is largely because users are relying increasingly on untrusted software for daily activities such as instant messaging communication and viewing various forms of multimedia content for business activities and entertainment. The high risk of damage posed by such applications has prompted research into practical techniques that can allow safe execution of untrusted software [28, 7, 24].

A majority of efforts in untrusted software security have been focussed on stand-alone applications using techniques such as policy-based confinement (also known as sandboxing) [28, 7, 24] and isolated execution environments (e.g., using virtual machines). However, these techniques do not address the serious threat posed by *software extensions* in plug-in and module-based software architectures. Examples of such extensions include browser plug-ins for viewing (or listening to) various forms of content on the Internet [1], Search or other toolbars for browsers, audio and video codecs, Apache modules[2], Office add-ons, and Photoshop image processing filters. More generally, software libraries such as those for image decoding and internationalization font support are instances of application extensions that operate with almost all GUI desktop applications.

Software extensions can alter application behavior in malicious ways because they reside within the same address-space as their host applications, and communicate with it via expressive APIs that have been custom-designed for the application. This enables attacks wherein the extension subverts the host application's logic by corrupting its data structures. Possible targets of such corruption include function pointers used by the host, variables holding the names of files written or executed by a program, buffers holding the data to be written by a program, etc. The threat is realistic and growing sharply, as confirmed by [30] which details that nearly 75% of the spyware on the Internet, uses browser extensions to monitor user activity. The attacks that specially crafted malicious extensions can perpetrate are much more powerful that those that exploits benign extensions. There is already compelling evidence for the latter, as seen in attacks targeting helper codec libraries[4], and multimedia players [3]. Clearly, there is a need to study the impact of using software extensions obtained from untrusted third-parties on the security of host systems, and develop corresponding defenses against the threat.

Static detection of malicious behavior is often hard because malicious code may use various forms of obfuscation to evade detection [19, 13]. In general, techniques that aim to detect a specific "signature" behavior have limited applicability as a proactive defense technique, because they give no guarantees against an attack. Software-based fault isolation (SFI) [29] is a language-based mechanism that provides such comprehensive protection at a lower cost than what is possible with OS-provided memory protection mechanisms. XFI [15] is a recently developed technique that can limit memory accesses made by extensions to specified ranges of addresses, and ensure that

the extension can call only a specified set of functions provided by its host application. These specifications essentially correspond to security policies that are enforced on extensions. The low-level nature of these specifications is motivated by the low-level nature of attack capabilities that extension code poses. Although these techniques are able to provide a possible foundation for securing extensions, their mechanisms are not easily adapted to achieve typical security objectives relevant to the extensions outlined earlier. There are two main reasons for this :

- Shared-memory plugins are popular because they can exchange complex data structures without worrying about where and how its components are created. As a result, host-extension interfaces tend to consist of aggregate data structures, which in turn contain multiple level of pointers to memory regions dynamically created by both the extension and host. Inferring memory access constraints becomes impractical in such a setting. This is because it is hard to identify all the memory regions that are granted legitimate access by the host application, out of the complete set of memory locations accessible through the pointers explicitly passed as function arguments to plug-in functions, or implicitly shared in the process address space. Moreover, if the host application changes some of the pointer values, that may implicitly change the memory regions accessible to the plug-in. Finally, the efficiency of memory protection using above mentioned techniques can degrade quickly if the number of memory regions increases beyond a few.

- The APIs exposed to the extension by the host typically consists of hundreds to thousands of functions. As an example, the KDE libraries, which provides the basic I/O and GUI services to most plug-ins (as well as stand-alone applications) in the KDE environment, contain over 24,000 functions. Given the large API being exposed to an untrusted plug-in, it is unreasonable to assume that these API functions are written carefully enough to protect themselves from maliciously crafted data that may be passed into them. As a result, policies that simply allow (or prohibit) access to a set of functions are not powerful enough to offer protection from plug-ins that use large host APIs.

This thesis aims to address these drawbacks in the context of complex extensions, using a secure *attribution* mechanism to identify which actions are performed by the untrusted plugin code, or by trusted host application on its behalf. In effect, this allows us to *separate the policy enforcement from the complex extension APIs and memory access locations, to other smaller APIs that can be secured easily,* such as the system-call API. The primary difficulty in delayed policy enforcement is that security-sensitive API functions may be invoked by the trusted host application as well as the untrusted extension. Our approach is to use information flow tracking to attribute the call to trusted (or untrusted code), and enforce appropriate security policies based on which object "controls" the action. The main advantage is that policies developed in system call based sand-boxing approaches [24, 27] for stand-alone applications, can be applied more or less directly for extensions providing similar functionality. Further, we discuss the threat model and possible attacks that an adversarial extension may perpetrate, and show how our approach is designed to defend against them.

In order to make the mechanism practical, we developed a static binary transformation and analysis tool for Linux/x86, that has a much more general applicability in security applications. It can be used as a whole-program analysis tool for binaries, and for robust heavy-weight transformation of large applications. In general , the thesis makes two main contributions to the area of program transformation :

- It provides evidence for the potential of combining static analysis and runtime enforcement on binaries as an effective technique, yielding performance comparable to those offered by similar approaches on source code.

- It demonstrates the relative merits of static binary transformation against dynamic translation, as a means to achieve more practical performance without sacrificing robustness.

In particular, we apply static analysis to recover some of high-level program structure unavailable in commercial COTS binaries. This enables several performance optimizations to dynamic taint-tracking instrumentation. The net effect of these optimizations is that taint-tracking on binaries incurs performance penalties that compete with those obtained by source/compiler transformations, and are better than those offered by contemporary binary based approaches by a factor of 4.

1.1 Organization of The Thesis

In Chapter 2 we describe the threat model imposed by untrusted software extension operating in the address space of trusted host, and the corresponding defense strategy. Chapter 3 describes the design and services offered by BinStat – a static analysis and transformation tool developed for realizing the system. Chapter 4 describes the performance optimizations developed in the context of binary taint-tracking using BinStat. In Chapter5, BinStat implementation and the implementation of other tools employed in sand-boxing are detailed. Chapter 6 presents and experimental evaluation the effectiveness of the defense technique, the robustness of the tool, and the performance gains achieved by the optimization techniques.

Chapter 2

Containment of Shared-Memory Extensions

Module and plug-in based architectures are becoming increasingly popular because of the flexibility they offer for extending the functionality of host applications such as browsers, web servers, email clients, office tools, and so on. The added flexibility comes at the expense of possible security compromises as a result of the excessive permissibility granted to untrusted components. Extensions such as browser plug-ins can access all of the memory of their host application. In addition, they can interact with their host application using application-specific as well as system-wide APIs consisting of thousands of functions. These factors make it very difficult to distinguish the actions of the extension from its host application, thus complicating efforts to selectively contain such extensions.

We could treat any application containing an untrusted extension as untrusted, and apply the techniques developed previously for securing untrusted applications. However, such an approach would typically be unduly conservative, precluding some of the intended uses of the host application. For instance, any addition of an untrusted plugin to a browser will render the whole browser untrusted, making it unsuitable for high-security operations (e.g., accessing an online bank). It is therefore necessary to develop techniques that can confine the extension code without limiting the privileges granted to the (trusted) host application.

Malicious extensions sharing the address space with host applications has serious consequences. They may be specially crafted to violate common conventions implicit in the host application code, which is typically compiled by a benign compiler. To understand the scale of possible damage, we first describe the threat model in Section 2.1. In section 2.2 we describe how previous approaches are either inapplicable or have dealt with this threat only partially ignoring important possible attacks . In section 2.3 a basic defense strategy is described, and analysis of its effectiveness against the threat model is presented. The implementation and complete experimental evaluation is deferred until later in the thesis.

2.1 Threat Model and Attacks

In this section, we show that adding a untrusted software extension that resides as part of a trusted host application, poses a powerful adversarial threat. Consider the simple case of multimedia decoder for a certain digital format of video. Typically, these “codecs” are available for downloads these from untrusted sources such as the Internet, and perform specific decoding operations on buffers explicitly given to it by a media player. Thus, such an extension is tightly coupled with the media player and is loaded dynamically as a shared library having full access to the process

address space of the media player. Moreover, for most media players (such as VLC player, Gxine, on Linux) there are equivalent software extensions available as shared libraries, that plug into a browser for viewing online videos. As a result, the security of the browser may be impacted by the loaded media player libraries, and the associated codecs it imports in the address space. Malicious media players or its codecs may directly perpetrate attacks on the browser.

We assume that these untrusted components are available in binary form. Even if their source code is available, it can not be trusted to have the same behavior as its binary as it may be compiled by a malicious compiler. Further, we assume there exists a *extension API* and a low-level platform defined *Application Binary Interface* or *ABI* for communication and inter-operability with the host. However, the untrusted extension may not conform to either of these. We restrict the untrusted extension to operate in user space, and not use vulnerabilities in the operating system for attacks. The end goal of the attacker is execute a sensitive operation such as a system call of its choice, or *"control"* the operation without alerting the user by causing a system crash.

We loosely measure the power or capabilities of an attacker, based on the control it is able to exert on *the choice of the sensitive operation*, and *the values of the arguments*. Clearly, the attacker that has a very limited choice for arguments to a given system call, say `read`, (such as static strings used the host application), is much weaker than an attacker having arbitrary control to execute any system call with any arguments. Consequently, there is spectrum of attacks vectors possible when varying the range from "no control" to "arbitrary control" for each of the mentioned parameters. We see here that there is inherent degree of freedom in defining the attack model. Later in our defense evaluation, we argue that even though our defense may not prevent an arbitrary attacker, it does render most of the practical attacks impossible.

In practical applications, we identify the following ways in which the attacker may compromise the system :

1. **Subverting program flow**. The program may subvert program flow of the trusted code by corrupting function pointers, and return addresses to directly execute sensitive operations or execute host code that in turn calls sensitive operations. Further, corruption of its own code pointers and return address may allow the untrusted extension to inject new code or execute code that may be hidden from disassemblers by obfuscation. Extensions may also use system features such as signals, exceptions, `setjmp/longjmp` to transfer control to unintended code.

2. **Violating program data integrity**. An attacker may directly corrupt data stored in the process memory of the process to control the arguments. It may exploit several levels of pointers to corrupt the intended target data in indirect memory accesses. Typically, this may be possible because most legacy plugin APIs[1] such as Netscape API are designed for benign plugins. Thus, the exchanged interface data structures contain objects owned by the host application as well as extension specific data. The integrity of host-owned objects are left unchecked at the interface, since plugins are assumed to return them unmodified. The attacker may also use system implementation knowledge of how threads are implemented to corrupt data. For example, on recent Linux versions, each thread is given a new base stack address, expecting that the stack based accesses will not overflow into thread stacks of other threads. However, this may be violated by the attacker to corrupt data on other thread stacks.

3. **Low-level attacks on ABI and violation the API**. An API usually defines which functions are legitimately accessible, and may be violated by extensions by calling prohibited functions. On the other hand, an ABI specifies certain low-level restrictions on the usage of registers and calling conventions. For example, it specifies that certain registers (called

"callee saved registers") are left unmodified across a function call, and specify which registers contain return values. Similarly, the ABI may specify that the value of the stack pointer register (`ESP` on x86), is unchanged after executing a function call. To enforce these, compilers generate code to save/restore caller saved registers and `ESP` in the callee functions. A malicious extension function may intentionally violate these conventions to subvert control flow, or pass malicious parameters to the trusted caller functions.

4. **Exploiting concurrency**. Certain attacks become possible in multi-threading applications, due to the inherent race conditions in data access between co-operating threads. Benign host threads are often designed to employ appropriate synchronization mechanisms to update shared data. However, interleaving of a malicious thread execution with benign threads may allow it to unduly race with benign threads and corrupt shared data structures. This in turn may lead to corrupted pointer usage by benign threads, and subvert data and control flow integrity.

5. **Attacks against the defense mechanism**. A malicious extension may directly corrupt the data state used by runtime monitor engine. Specially for instrumentation based approaches such as SFI[29], there is a threat of malicious code bypassing security checks by jumping across them. In addition, approaches that track metadata such as taint-tracking [23], suffer from the possibility of exploiting concurrency of data and metadata access, to cause the metadata to be out of sync with the corresponding data. All of these could be exploited to evade runtime detection.

6. **Subverting program logic**. Program logic can be extremely complicated and creates dependencies between data variables in the program. There may be direct data dependence between data that is directed copied from one variable to another, or updated through pointers. Other dependencies like control dependencies and implicit information flow may be exploited to evade detection.

   An extension may exploit various kinds of dependencies inherent in the program logic. If the defense is insecure, it may craft its code to use these dependencies to corrupt variables that eventually affect the target data, without being detected. Finally, it is possible that it may find such dependences in the host code and modify an seemingly unimportant variable to have the host application perform an unintended action. We discuss these in more detail later in the chapter.

2.2 Limitations of Previous Approaches

Previous works have proposed to address this problem by limiting memory sharing, and by using small security-aware APIs. However, some of these techniques, like XFI [15, 29] are fault isolation techniques designed to limit the damage of programming errors in extensions, and hence do not defend against malicious extensions. As a result, these techniques are not designed to consider all the possible attacks outlined.

XFI limits the set of addresses that the extension has legitimate access to, and prohibits it from calling functions that are outside the API. Extending the basic XFI technique to deal with powerful extensions such as flash players and document viewer libraries used in browsers, has practical limitations. Host-extension APIs tend to be large and involve complicated data structures to enhance programmability and flexibility. For instance, a media player may use a number of host API functions: a network API for communicating with an Internet site originating a media stream,

a window API for displaying video content and user interaction, file API for storing temporary files (or preferences files), and so on. As a result, even after limiting accesses to certain functions, it becomes far too cumbersome to sift through such large APIs to identify which functions can be safely exposed to untrusted extensions, and if so, with which parameter combinations. It is reasonable to expect that these libraries designed for non-malicious clients do not have comprehensive input validation checks that would be needed at a security-sensitive interface. Besides, extensions can implicitly communicate with host application by simply modifying shared data structures *without making any API calls.*.

To tackle these problems, sharing may be restricted, but such restrictions typically negate the main benefits of shared memory. For instance, if we restrict sharing by predefining the address ranges that can be shared, we complicate programming by requiring data to be copied back and forth into these regions before each API call and/or return. Worse, one has to decide ahead of time which pointers in a linked data structure would need to be accessed during a call, and the targets of all such pointers would need to be copied. This can be expensive in terms of performance, as well as the programming effort required to identify the regions to be copied and to write the code to do the same. Such copying may even break the semantics of shared mutable data structures and leave the application unusable.

A different approach to the problem of detection malicious behavior at runtime, is taken in [14]. This technique may be effective as a technique to detect malware that is currently available, but is not directly applicable as proactive containment mechanism to deal with malware specifically crafted to evade detection. As a simple example, consider the malicious extension that leaks sensitive data by using covert channels like implicit flows. Besides, to build any confidentiality policy there should be a basis to ensure low-level integrity to build on. This integrity seems to be assumed in their work.

2.3 Policy Enforcement Based on Secure Attribution of Actions

Our approach is based on the intuition that unlike the APIs exposed to extensions, which have flexibility as the primary objective, security-oriented APIs tend to be smaller, and carefully control how shared memory is used for communicating data cross the API. For instance, the system-call API on Linux consists of a couple of hundred functions rather than hundreds of thousands. Moreover, the kernel code that implements system calls is memory protected from application code that invokes these calls. It is much easier to define and enforce security policies with such APIs, as evidenced by the rich variety of practical policy enforcement tools[24] based on this API.

In contrast to memory region based confinement approaches, we develop an approach that enables securing extensions that rely on complex APIs. Our approach achieves this by moving policy enforcement from the extension API to other APIs that can be secured easily, such as the system-call API. Instead of eagerly enforcing security policies at the bulky host-extension interface and memory access locations in the extension, enforcement is delayed until control-flow reaches a system call. At this interface, security properties of interest can be easily expressed and safely enforced.

Security-sensitive API functions may be invoked by the trusted host application as well as the untrusted extension. In addition, extension code may call browser callback functions to perform its intended action, or may simply modify the data used in arguments to system calls. Thus, our approach tries to identify the *"control context"* in which a system call is made. Specifically, we use *secure attribution* of operations, to attribute system calls to one of the three contexts : (a) plug-in, (b) host application, or (c) a host-application function called by a plug-in. Different *system-call*

based sandboxing policies are enforced based on this attribution, thereby enabling plug-in operations to be sandboxed without having to restrict operations being performed by the host. This technique has the advantage that it simplifies policy development, ensuring that certain high-level security objectives are achieved without having to expend enormous efforts in developing or customizing policies for each specific plug-in or module. Policies already developed in previous works [24, 27] for stand-alone application , are shown to be applicable with little or no modifications to selective contain extensions offering similar functionality.

This problem of identifying contexts in which function calls are invoked has been investigated by researchers in past. Anomaly detection techniques such as those which use return addresses from the stack[17]. The detection mechanisms employed are based on some benign assumptions based on the structure of the call stack and data integrity at the point of invoking system calls. Thus, the attribution mechanism is insecure and attackers can spoof the return address information to evade detection. Unlike techniques such as these, stack inspection capabilities are built into the the design of certain higher level languages such as Java. This has been used for inline reference monitoring[16] by certain tools. Our technique provides this context information *in an in-circumventable secure way on binaries*. It uses program instrumentation and protection of monitor state to achieve this.

The primary basis to track which data and function pointers are controlled by the plugin is *information flow tracking* or taint-tracking [23]. Different transformations are used for trusted and untrusted code. For untrusted code, the transformation is very simple: every byte of data written by it is marked as tainted. Trusted code is transformed to perform taint-propagation. In particular, the result of an arithmetic or logical operation is tainted if any of the operands are tainted. Similarly, the result of an assignment is tagged as tainted if the right-hand side expression is tainted. By propagating the "taint" information for all data in memory, we effectively identify which data is controlled by the untrusted code. Similarly, we identify all indirect control transfers made through tainted pointers (including return addresses), as belonging to the context of the untrusted code. In addition to providing control context information, taint propagation provides a secondary line of defense for checking integrity, giving fundamental low-level guarantees to enforce higher level policies on it.

Moreover, this approach requires no require any access to the host or plug-in source code. Instead, it is implemented by rewriting their binaries. In the chapter 6, we show the robustness of the approach to deal with real applications, and how the performance overhead of the whole approach is kept below 100% by means of static analysis based optimizations. Our experimental results improve upon previously reported binary based taint-tracking approaches [25] by a factor of 4.

2.3.1 Policy Enforcement

Using fine-grained taint information, we can now support a wide range of security policies for confining untrusted extensions, while providing roughly the same level of flexibility, power, and ease of policy development as with previous works on securing stand-alone untrusted applications. Specifically, the following kinds of policies can be supported.

- **System-call based sandboxing**. A system call can be attributed to an extension if the current control-flow is tainted, or if its arguments are tainted. No policies are enforced on system calls attributed to the trusted host application. For system calls attributed to the extension, we enforce a sandboxing policy that is appropriate for a stand-alone application providing the same functionality. For example, a document viewer plugin may allowed to display a certain temporary file downloaded by the browser on user consent, but prevented from performing any network communication. By further limiting any tainted data from

being written out through network write system call, we can prevent leakage of sensitive user data in the browser directly to a remote attacker, through the actions of the plugin.

In general, this basic approach can be further refined, e.g., to ignore taint on system-call arguments that don't significantly impact security. Moreover, if some extension API functions are known to incorporate adequate input validation, then we can untaint the arguments to such functions at the point of call. This is often referred to as *endorsement*. Finally, we may introduce finer granularity in attribution, e.g., distinguish between system calls made directly by an extension from those that are made by the host application on behalf of the extension.

- **Information-flow based OS Integrity**. There has been a resurgence of interest in information flow based mandatory access control (e.g., in Windows Vista) due to its potential for ensuring OS integrity in the face of an alarming rise in malware. These techniques are currently applicable to stand-alone applications. They tag any file created by an untrusted application as having low-integrity, while files created by trusted applications are tagged as having high-integrity; and impose a policy that prevents untrusted applications from writing to high-integrity files. Using our approach, these techniques can now be extended to work with shared-memory extensions. Specifically, if a file write operation is attributed to an extension, then the file can be marked as having low integrity, while files modified only by system calls attributed to the host application may be marked as having high integrity. We can also enforce a policy that prevents system calls attributed to the extension from modifying any high-integrity files.

- **Confidentiality policies**. For stand-alone untrusted applications, confidentiality is achieved by (a) preventing writes to the network (or public files, i.e., files that may be read subsequently by unauthorized principals), or (b) preventing reads from files that contain confidential data. Using our approach, both these policies can now be enforced on extensions. To achieve (a), we simply enforce a policy that prevents tainted data from being written to network or public files.

  To achieve (b), we need enhanced information flow tracking that keeps track of confidentiality in addition to taint. Since our taint-tracking implementation is capable of maintaining 8-bits of taint per (32-bit) word, it is easy to designate one of these bits to store confidentiality information. Information to be kept confidential from a plug-in, such as data read from sensitive files or sensitive data received over the network, is marked as confidential. Finally, a policy is enforced that prevents untrusted extension (or any code executing on behalf of it) from accessing data marked as confidential.

- **Untrusted host applications with trusted extensions**. It is often desirable to provide additional privileges to untrusted applications in a controlled way. An untrusted program may be permitted to overwrite a local file if the user consents to this. This consent may be obtained through a trusted library that performs appropriate checks and interacts with the user to get her permission. To illustrate this idea, consider a web server that hosts public web pages, in addition to password-protected pages that are accessed using SSL. We may want to restrict the access to these password-protected documents only if the access is performed through the web server's SSL module.

  As another example of a trusted extension, consider an untrusted application for which we want to provide very restricted network access, e.g., a plug-in that should be permitted to communicate with the domain that provided its content, but not others. Note that such a plug-in may still need to do a host name lookup. We could permit this by stating that it

can call a trusted library function `gethostbyname` which will in turn initiate the necessary network communication to perform the lookup, rather than trying to write a low-level policy that permits access to any DNS server in the world.

Our attribution-based approach can naturally handle trusted extensions. When an extension function is invoked by an untrusted host application, the arguments are untainted, and a control-flow context flag is set to indicate that execution is no longer under the control of the untrusted host. System calls that result due to the invocation of this trusted function will now be attributed to the trusted extension, and hence will be permitted. However, system calls made directly by the host application (or using extension functions that are not trusted in this way) will be attributed to the untrusted host application and appropriate policies will be enforced.

Delayed Vs Eager Enforcement The primary benefit of delayed enforcement (embodied in our approach) is that it simplifies security policy development, and allows more applications to execute successfully. Otherwise, we would be forced to write policies on safe access to a large number of functions, and moreover, be required to decide safety right at the point of invocation of each such function. In addition, our approach makes a more realistic expectation on the trusted application code — rather than assuming that every host application function that is callable by an extension can protect itself from malformed input data, it provides a uses taint propagation as a foundation defense technique. This allows policies of various kinds to prevent control or data subversion of the host application functions.

The drawback of delayed enforcement is that it becomes possible for a malicious (or buggy) extension to corrupt or confuse its host application to the point that it crashes, or simply does not perform its function as intended. However, this would usually lead to a system crash and would alert the user. To minimize such effects, any policies that can be easily specified and safely enforced at the extension API should be enforced, e.g., we may prevent an extension from writing to the stack memory used by thread(s) executing trusted code.

2.3.2 Instrumentation for Taint-Tracking

We maintain the taint information in an array *tagmap*. For a location l, $tagmap[l]$ indicates if this location is tainted or not. Tag space could be allocated statically, or using an on-demand allocation as in [31]. We associate 8 bits of taint with each 32-bit word. This sacrifices some accuracy in taint-tracking for byte arrays, as the same taint byte is used for all 4 bytes in a word, but this doesn't raise any significant issues in our application. Availability of 8-bits of taint means that we can capture (a) multiple levels of data integrity, (b) track information flow from multiple sources, or (c) track data confidentiality in addition to integrity.

In addition to memory, taint bits need to be maintained for each register. For the purposes of this discussion, it is useful to think of this data as being stored in *virtual registers*. In the code snippets in Figure 2.1, we use a virtual register `r_t` to store the taint associated with a CPU register `r`. Additional virtual registers `VR1` through `VR3` will be used for address computation (i.e., computing the location of $tagmap[l]$ from l) and taint-tag computation. Since virtual registers will ultimately be realized using memory, the instrumentation shown in Figure 2.1 uses them like a memory operand rather than a register operand. Since registers are part of thread-specific processor state (which is saved and restored by the OS/thread libraries on each thread switch), to preserve the semantics of virtual registers, they need to be stored in storage locations that are unique to each thread. On Linux/x86, thread-specific data pointer is accessed using the GS segment register.

```
mov eax, VR1
mov ecx, VR2
lahf
mov eax, VR3
lea [ebp+0x1c], eax
shr 0x2, eax
mov ebp_t, cl
or [eax+tagmap], cl
or ebx_t, cl
mov cl, [eax+tagmap]
mov VR3, eax
sahf
mov VR2, ecx
mov VR1, eax
add ebx, [ebp+0x1c]
```

(a)

```
mov eax, VR1

lahf
mov eax, VR3
lea [ebp+0x1c], eax
shr 0x2, eax



mov 0x1, [eax+tagmap]
mov VR3, eax
sahf

mov VR1, eax
add ebx, [ebp+0x1c]
```

(b)

Figure 2.1: Instrumentation for an `add` in (a) trusted code and in (b) untrusted code.

Specifically, we use an offset from the pointer at memory `gs:0x0`, to point to thread-specific storage where virtual registers are stored.

Figure 2.1 shows the basic taint-tracking instrumentation for trusted and untrusted code for an `add` instruction that adds the `ebx` register to memory location `ebx+0x1c`, leaving the result in the memory. The first step is to save `eax` and `ecx` registers so that they could be used in the instrumentation. Next, we save the CPU condition flags in `eax` so that they aren't clobbered by the newly introduced taint-related computations. (We use `sahf`/`lahf` instruction combination, which moves the flags into or out of `ah` register, instead of the more expensive `pushf`/`popf` sequence [25].) The flags are then moved into `VR3` to free `eax` for address computation, i.e., to compute the address where the taint tags of the memory operand are located. In the trusted code, which does taint propagation, we treat the data accessed using a pointer to be tainted if the pointer itself is tainted. This is why `cl`, which is used to compute the taint tag of the result of the add operation, is initialized with the tag of the pointer `ebp`. Next, we compute the logical "or" of this value with the taint of the two operands to `add`. The result is then stored as the taint of the destination operand, `[ebp+0x1c]`. Finally, the original values of the flags and registers are restored, and the original `add` instruction is inserted into the instrumented code.

Note that constants have a taint tag of zero, and hence binary operations that have a constant operand need not update the taint tag at all. A few exceptions that require special handling are instruction patterns of "xor reg, reg" or "sub reg, reg" which are pervasively used to clear a register, complex instructions such as string instructions which logically implement the semantics of more than one basic instruction, and instructions that have implicit operands such as "leave".

Instrumentation for untrusted code differs in one important way: any write by the untrusted code causes the corresponding taint tag to be set, so we move the constant "1" into tagmap instead of performing any computations on operand taint tags. This simplification (over trusted code) avoids the need for a few additional save/restores of registers, thus decreasing the instrumentation size significantly.

Figure 2.2: Scheme showing the points at which context flag bits are set and reset in a trusted function `BenignFunc` and an untrusted function `UntrustedFunc` . If only C_t is set, the program is in trusted context. If only C_u is set, the program is in untrusted context. If both bits are set, the program runs in trusted context on behalf of the untrusted code.

2.3.3 Instrumentation for Attribution

Attribution of sensitive operations (such as system calls) is based on two factors: the control-flow context in which the operation is invoked, and the taint information associated with its parameters. This section describes how control-flow context is attributed.

At any given instant, a thread of program execution could be in one of three control-flow contexts: (a) untrusted code (b) trusted code, and (c) trusted code called on behalf of the untrusted code. Our instrumentation uses two context flags C_t and C_u to track these contexts. C_t is set whenever execution is within the body of a trusted function, while C_u is set when there is a untrusted function active anywhere on the current call stack for a given thread. Specifically:

- C_t is set at the beginning of each trusted function, and immediately after any `call` in its body. It is reset at the beginning of every untrusted function. It is also reset immediately following all calls within untrusted code.
- C_u is set at the beginning of each untrusted function and immediately following every `call` in its body. It is reset at the end of each untrusted function. C_u is also set whenever a call is made using a tainted pointer.

We point out that the instrumentation in Figure 2.2 are specifically designed to avoid knowing the call target address for computing the context flags. This is important because such address lookups are expensive at runtime, and may add unnecessary complexity in the system.

2.3.4 Integrity of Program Flow and semantics of the ABI

To ensure *control flow integrity* [6], we first perform a static inspection of the untrusted code during disassembly to ensure that the program is assembled in a benign way. In this phase, we check that no static intra-procedural jumps cause program execution to continue in the middle of instructions. All static calls are checked to be to a valid entry point in the function. In case of optimizations, such as tail-call elimination, functions may not return to their caller but instead to a sibling function. These transfers are statically determinable and explicitly allowed. The code is checked to confirm to alignment restrictions of the ABI, such as all function starts being 4-byte aligned. No control transfers are allowed to be made outside the disassembled code section of the executable. In shared libraries on Linux, external control transfers are limited through a procedure linkage table (PLT)

mechanism which uses a function pointer dynamically updated by the runtime linker. These calls are statically identified and appropriate function pointer integrity is maintained by checking the taint information with them at runtime.

Runtime checking is needed for all indirect calls and returns from functions in untrusted code. Control is allowed to follow to valid function entry points and return targets respectively. To implement this, we associate tag bits with each valid function start and return point. In our approach, all valid function starts and return points are given the same value. Tag values are not stored inline in the code as in [6], but in a separate read-only array of bits, added statically as a section in the executable binary. Indirect jumps, such as those typically generated in jump tables, are checked at runtime to transfer control to valid basic block starts.

For the trusted code, control flow integrity property is assumed applied by the compiler. Only the runtime checking of indirect control transfers, including returns from functions, is performed to ensure that these code pointers are unmodified by untrusted code. Specifically, we check if the associated taint information indicates that they are untainted. Control transfers through tainted pointers cause the C_u flag to be dynamically set, indicating that the untrusted code controls the execution context. Appropriate policies for untrusted context become active from this point.

Semantics of the ABI are specifically enforced on control transfers that cross the host-extension interface. For example, on Linux/x86 most ABI require that callee-saved registers and `ESP` are unmodified across a call. This may be enforced by implementing a protected shadow stack, where these values are pushed upon the cross-domain control transfer and checked upon return. Like all other taint information, the protected shadow stack is prevented from corruption.

2.4 Effectiveness of the Defense

In this section, we argue that this defense forms a secure basis for enforcing policies at security APIs such as system call API. The first observation is the mechanism allows us to determine the context of a function call by using the context information encoded in the C_u and C_t flags. Note that the plugin can not stray outside the API without causing the control context to switch to untrusted. Indirect control transfers and direct external calls appropriately set the context to untrusted when the code pointers are tainted. As a result, attacks such as function pointer corruption are rendered ineffective, as the attacker achieves nothing more than it would by calling the function directly. Moreover, this approach is more realizable in practice because it implies no knowledge of which host functions are callable by the extension. Indeed, if this information were made available or inferred by other means, additional restrictions can be easily integrated in our system by using control transfer tag checking as detailed earlier.

Control flow subversion is prevented in our approach by enforcement of control flow integrity in untrusted code, and by runtime checking of code pointer integrity based on taint information. Our mechanism never allows execution of code undiscovered at disassembly. This deters the attacker that aims to achieve a significant objective from using obfuscation to evade disassembly, since execution of undiscovered or injected code triggers an alert. Signals and other exceptional flows that originate in the untrusted code get automatically handled, as the C_u will be set.

All data written by the untrusted code is tainted. In addition, all static data in the extension, which is typically a shared library, is initialized as tainted. Thus, taint information for arguments at sensitive API functions, allows us to detect which arguments may be under attacker control. Further, whenever the pointer of a data access is tainted, the corresponding data write is also marked tainted. This mitigates the threat of corrupting higher level pointers that eventually leads to malicious data values.

Our technique is designed to defend against attacks that target our specific technique by corrupting any instrumentation data. The most obvious target in this context is the tagmap. Untrusted code may attempt to overwrite the tagmap directly, or by exploiting memory errors in trusted code. This attack is prevented using the technique described in [31]. In particular, note that any instructions in (trusted or untrusted code) that writes into a memory location m within the tagmap will be preceded by an instruction that updates $tagmap[m]$. By leaving $tagmap[l]$ unmapped for all locations l that fall within the *tagmap* array, such an instruction will cause a memory exception, causing the program to be aborted. Note that this technique can be extended to protect all data structures that are used by our instrumentation since none of that data needs to be accessed by the original code. Thus, other data used by the instrumentation, including (a) memory allocated for virtual registers, (b) data structures used to represent allowable jump or call targets, and (c) data structures used for policy-checking, (d) data that ensure ABI semantics at the host-extension interface such as the shadow stack, can also be protected from corruption.

Finally, we address the threat of exploiting the program logic and concurrency inherent in multi-threaded applications, to propagate untrusted data without propagating corresponding taint. First, we point out the malicious code can not achieve this directly by using its own program logic, since all data written by it is marked tainted. This addresses the use of all covert channels such as implicit flows in specially crafted malware to corrupt program variables without propagating taint. We argue that tracking data dependences through pointers in the trusted code, in addition to this kind of marking in untrusted code significantly weakens attack capabilities in practice. For the sake of discussion, consider the case of an attacker that intends to write into a trusted file, such as a file maintained by the browser to store persistent cookie information of the user. Realistically, to achieve this the attacker should be able to control (i.e., choose any arbitrary value for) either the file pointer , or the data being written to the file write operation. One slightly subtle way the attacker may achieve this is by corrupting pointers used by trusted code, to trick it to inadvertently perform the operation for the attacker. However, our tracking of such pointer corruption will result in the data accessed by the pointer to be tainted, thus defeating the attacker objective. As can be seen, attackers that enjoy unbounded freedom in today's systems to arbitrarily corrupt data and function pointers, are highly limited in the control they exert in presence of this technique. We point out that attackers have to find existing indirect (or control) dependences and covert channels in benign host programs, to carry out their exploits. These channels are not considered a significant practical threat as evidenced in most prior works on taint tracking [31, 23], that do not track control dependence and implicit flow in benign code.

To complete the discussion of ways in which attacks can cause our tracking to miss propagating corresponding taint, we consider concurrency attacks. The problem is simple - taint information and data access operations are not performed atomically. This could result in two kinds of races – write-write and read-write races. We do not worry about races where untainted data is written whereas the corresponding taint tag ends up being "unsafe". Such as case would only potentially flag safe data as unsafe leading to false triggers, but can not be used by the attacker to corrupt data without being detected. The problematic case is the reverse – at a policy enforcement point the data is marked with "safe" taint, whereas it actually contains "unsafe" values controlled by the untrusted code. In subsequent discussion, this dangerous condition is referred to as DC, and must be carefully avoided in the design. This problem could be addressed by using appropriate synchronization mechanisms in the taint monitor code to ensure that data access and taint tag accesses happen atomically. Practically, performing lock synchronizations at instruction granularity will lead to unacceptable performance. Fortunately, on the x86 architecture there is surprisingly efficient way to prevent this dangerous condition from occurring.

For the purpose of discussion, let us assume that memory load/store operations in the target instruction set either write a memory location using a register, or read the contents of a memory in a register. The main design idea is that in the untrusted code, we perform taint marking before as well as after the data access, whereas for the trusted code, we reverse the order of taint and data access based on whether the instruction is a memory load or a memory write operation. In trusted code, taint access follows the data access for memory load instructions, and the exact reverse for memory write instructions. The justification for this technique is explained next, and is based on the assumption that no two benign threads race with each other for a data access. This is a reasonable claim : correct multi-threaded program threads are likely to employ appropriate locking mechanisms to ensure mutual exclusion in accessing shared data. The only problem is when a malicious thread tries to interfere with the data access with other benign threads. There are two kinds of races possible :

- **Write-Write races**. In this case, both the benign thread B, and the untrusted thread U, write to the same data. If both threads propagate "unsafe" or "1" taint, then there is no issue. However, if B writes a "0", it intends to write benign data in the location where U writes untrusted data simultaneously. To prevent condition DC from occurring, the taint write operation in B happens before the data write, whereas the opposite happens for U in our scheme. Even with arbitrary interleaving of the 2 operations in B and U, it is simple to check that DC becomes impossible.

- **Read-Write races**. The untrusted thread U performs three operations after instrumentation - writes a "1" taint denoted by operation $T1_u$, writes unsafe data denoted by D_u and finally writes taint again denoted by operation $T2_u$. The benign thread B reads data in operation R_t, and then reads the associated taint in operation T_t. For condition DC to occur given the ordering of taint reads and data reads, there are only two cases possible:

  (a) The benign thread reads tainted data, but before it reads the associated taint, the taint is cleared. This can happen only if the writing thread is a benign thread. This would be a race between two benign threads — a violation of our assumptions.

  (b) D_u precedes R_t, $T2_u$ follows T_t, and the taint value read in T_t is "0". This is not possible because operation $T1_t$ ensures that taint read in R_t is marked "1". Again, between $T1_t$ and T_t, the taint can not be cleared by another benign thread, as this is violation of our assumptions. Thus, condition DC does not occur.

Therefore, we reason that condition DC is never possible in our scheme. Our assumptions about architecture are largely true for the x86. On the x86, instructions can have at most one memory operand with some rare exceptions that need special handling such as string operations. All indirect memory references use registers as base and index, so there is still only one memory operand involved in each instruction. Other than the load/store operations, there are arithmetic instructions of the form `‘‘ x = x op y’’` which may read and write memory locations if `‘‘x’’` happens to be a memory location. However, in such a case `‘‘y’’` has to be a register which is a thread-specific data. We point out that our scheme naturally deals with this as no new race conditions are introduced beyond those outlined. This is because semantics for taint propagation for such instructions, requires to check the taint for the register operand `’’y’’` – iff the taint is `‘‘1’’` the corresponding taint operation is to store a `‘‘1’’` taint for `‘‘x’’`. Thus, this is exactly like the case of a conditional store instruction that equivalently "writes unsafe data in `x`". This is handled correctly in the scheme, and adds no special cases. We also point out that the basic technique scales to more than one benign thread operating alongside the untrusted thread, and leave out the rigorous argument for brevity.

Chapter 3

Static Binary Analysis and Rewriting

Program transformation has played a major role in enforcing various kinds of security policies on critical applications. Fine-grained system call based sandboxing, inline reference monitoring and fine-grained taint tracking are some examples of well-developed techniques that have heavily relied on program transformation. Much of these are based on source-source transformation or on instrumentation during compilation. Using tools such as [12, 21], previous works have shown that heavy-weight instrumentation such as taint-tracking is practical, even though the technique typically introduces more than one instrumented operation per source operation. When dealing with binaries, most systems today turn to dynamic instrumentation based infrastructures like DynamoRio[18, 20, 9] which defer the instrumentation to runtime when the code is first executed. However, such techniques suffer for extremely high performance overheads to be deployed in practical use. There is little evidence of robust static techniques for performing such fine-grained instrumentation on binaries. Vulcan is an infrastructure developed by Microsoft that is fairly robust, but relies on much symbolic and debugging information that is embedded in executables for its analysis. Similarly, Diablo is a similar tool for GCC based tool chain. However, relying on this information when dealing with untrusted code is questionable, and for most stripped applications on platforms like Linux such information is simply not available for third party code. Other techniques [22] are quasi-static as they combine runtime disassembly with static techniques, but there has been no evidence of heavy-weight instrumentation such as taint-tracking using these.

Working on binaries has some advantages over source code. Techniques developed to work on binary applications have direct applicability to a large set of programs. Programs may be written in different higher-level languages and compiled by different compilers, and yet can be transformed in a single framework. Machine code semantics offer a common platform and moreover, it serves a natural interface to address low-level attacks such as memory corruption.

Binary instrumentation has the unique application in security – it is often the only way to deal with third-party software for which source code is unavailable or untrusted. Even for benign programs such as web browsers, dealing with source code for transformation has important practical problems. This is largely because of following reasons :

- Large host systems have complex build processes. For example, compilation of many systems, generates source code on-the-fly during compilation based on templates. Understanding and modifying complex build systems of each application, to intercept all the generated code compilation can be a challenging and cumbersome experience.

- During compilation, programs are heavily optimized and additional code for implementing language features such as polymorphism and runtime type checking are introduced.

- High-level languages such as C allow programmers to write assembly code which is typically untransformed in compiler or source instrumentation. Besides, all code available as hand-written assembly is not amenable for these forms of transformation. Many popular web browsers use hand-written assembly in libraries used for providing OS and platform dependent services.

More generally, transformation as well as program analysis has been limited on source code by the problems of separate compilation and those mentioned above. This is because most production compilers today process one source unit at a time, such as a function or a file. These units are linked together to form a executable binary after compilation. As a result, most analysis that work on source are limited to function or file scope, rather than the whole-program. Link time transformation, besides being completely compiler (or tool-chain) dependent, can be as challenging as transforming binaries and have limited applicability because they rely on information derived from source such as static relocations. In this regard, analysis on binaries offers a much better option for several whole-program analysis and transformation of executables.

Robust transformation and scalable whole-program analysis has problems when dealing with raw binaries. First, static disassembly of stripped binaries is a hard problem. Several techniques have addressed these issues with static approaches [26] and quasi-static approaches [22, 5] and better techniques are being investigated in the research community. A second limitation of working with binaries is the lack of high-level program structure to perform various powerful static analysis. We address this limitation by developing techniques for recover some higher-level program structure and develop a new static alias analysis algorithm suitable for COTS binaries. Further, we show the effectiveness of performing static analysis based optimizations to lowering performance penalties by applying it in the context of fine-grained taint-tracking. In particular, we achieve performance overheads 4 times better than those developed by previous binary-based works.

In this chapter, we present the design of a static analysis and a robust transformation system called **BinStat**. BinStat is designed for ELF file format on Linux/x86 and performs analysis/transformation on both shared libraries and cooked executables. Due to the practical limitations of disassembly of stripped binaries, BinStat currently relies on symbolic information for the disassembly step. It does *not* use any symbolic or relocation information for any other analysis or transformation with the aim that once better disassembly techniques are developed, all techniques it employs will remain fully applicable to stripped COTS binaries.

3.1 Design

The basic design of the framework consists of the following components - a disassembly engine, a static analysis framework, and an instrumentation engine. For disassembly, we start with the assumption that the entry points of functions are identified. Next, we perform a recursive traversal based disassembly of the program. The disassembly engine relies on techniques used in [32]. The basic static analysis subsystem is described first, followed by the program transformation subsystem.

3.1.1 Challenges in Binary Analysis

Our approach is primarily targeted for binary code and utilizes sound static analysis for performing optimizations. There is relatively little evidence of sound sophisticated static analysis techniques employed for instrumentation at the binary level. This is largely because analyzing memory accesses has been hard on x86 binaries, specially in the absence of symbolic or debugging information. The main problems are due to indirect references which are impossible to reason about due to memory

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

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

struct ABC get_min (struct ABC* x) {
    struct ABC temp = minimum;
    if (compare_lesser_than (x, &minimum))
        temp = *x;
    return temp;
}

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

int compare_lesser_than (struct ABC* x, struct ABC* y) {
    return compare (x->a, y->a);
}

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

Figure 3.1: A simple C program that finds the minimum element in an array.

aliasing, in general. However, these problems exists at source code in higher level languages such as C and C++; yet compiler optimizations are able to improve performance by many factors. A deeper understanding reveals that a lot of this bias is attributed to the semantic gap between binaries and source code, such as the scope and type information associated with variables. In binaries, there is no notion of program variables, a very weak notion of types (associated with sizes of operands in instructions), and no information about scope.

However, most of the binary code used today is compiled by benign compilers. Even most CPU architecture designs mirror higher language designs, such as support for `call` instructions similar to a function call in higher level languages, and for implementing stack-based language runtime by providing special stack manipulation instructions that implicitly use `ESP`. Also, compilers have been much more successful reasoning about local variables since they can reason about their scope in a sound way. In the context of code instrumentation, many source based instrumentation techniques have relied on such optimizations in the past by "piggybacking" on standard compiler optimizations to achieve performance. For example, [31] introduces local taint variables for local program variables during source transformation, which later gets optimized by the compiler to get significant performance gains. Therefore, the focus in our analysis to recover enough information about memory accesses that correspond to "function local" variables in the programs. In fact, this is a reasonable strategy to adopt based on the practical observation in previous works [31] that programs do often use local variables for computation.

We begin by first demonstrating some of the challenges in binary analysis of x86 code using a contrived example in figure 3.1. The example captures the some of the features that is common in large applications written in languages in C/C++ that designed for extensibility, such as use of indirect function pointers, passing parameters by reference and use of aggregate structures. The

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

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

<compare_lesser_than> :
     push ebx
     sub 8, esp
     mov 20[esp], eax

     call get_pc_thunk_bx
     add STATIC_DATA_OFF_1, ebx

     mov [eax], eax
     mov eax, 4[esp]
     mov 16[esp], eax
     mov [eax], eax
     mov eax, [esp]
     mov offset_funcptr [ebx], eax

     call [eax]

     add 8, esp
     pop ebx
     ret
     .
```

```
<get_min>:
     sub 28, esp
     mov ebx, 12[esp]

     call get_pc_thunk_bx
     add STATIC_DATA_OFF_2, ebx
     ...
     mov ebp, 24[esp]
     mov 32[esp], ebp
     ...
     call compare_lesser_than
     ...
     mov esi, [ebp]
     mov ebp, eax
     mov 24[esp], ebp
     add 28, esp
     ret 4
```

```
<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 minimum_offset[ebx], ebp
     lea 4[eax], esi
     lea 40[eax], edi
.L11:  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 .L11
     pop ecx, ebx, esi, edi, ebp
     lea -4[ecx], esp
     ret
```

Figure 3.2: Binary code for the C program in Figure 3.1

corresponding assembly code is shown in Figure 3.2, when compiled for production use as a shared library using "gcc" and typical optimizations (with command line arguments `gcc-4.1 -O2 -fPIC -fomit-frame-pointer`).

The C program shown in Figure3.1 finds the smallest element in the array `array`. The elements of the array are structure aggregates, the comparison function for which is implemented by `compare_lesser_than` function. It calls a function `subtract` through a function pointer `compare` to perform its operation. The assembly code in the example is pruned for conciseness, and certain instructions are clubbed together to enhance readability. Using this as the primary example, we outline some of the difficulties with analyzing memory accessing in binary code.

First, the accesses to variables in the program are no longer explicit. Upon closer inspection it is seen that all static data references are resolved through a indirect pointer obtained by adding a constant value(`STATIC_DATA_OFF_1` in function `compare_lesser_than`). The base pointer is implicitly returned by the function `get_pc_thunk_bx` in the register `ebx`. However, this is one instance of the way that "gcc" generates these references, and several others exists. State-of-the-art analysis tools use pattern matching based approaches for detecting accesses to memory locations - for instance, IDAPro treats ESP and EBP based accesses to be stack accesses. These techniques may miss important cases or give spurious results which result in subsequent being unsound. Most local variable accesses use `ESP` as a base pointer, but not all. The function `main` uses the register `ECX` as a pointer to access local variable on the stack since the value of `ESP` may change during the initial stack alignment operations (to 16 byte boundary) in `main`. Typically, `EBP` is used as a frame pointer to access stack parameters as well, but can be eliminated for this purpose in production code as shown in this example. Therefore, the access to formal parameter for function `get_min` at this point would be missed without performing analysis. Therefore, recovering some notion of local variable accesses in a sound way without missing any cases is a hard problem.

A second difficulty is that information about returned and formal parameters of a function is lost after compilation. It is not apparent without analysis that instructions X and Y push parameters to function `get_min` on stack, while instruction Z does not store an actual parameter. Return values are usually returned in `EAX` as specified by the ABI, but other values of interest may be returned *implicitly* in other registers, such as the pointer implicitly returned in register `EBX` by the `get_pc_thunk_bx` function. Furthermore, source code based approaches can perform aggressive optimizations since they reason about the effects of executing a function on local variables. For example, the source transformer can ascertain that the call to `get_min` in `main` modifies no local variables, and thus can allocate local storage for the loop counter `i` in `main`. Note that, as an effect of further optimizations in the compiler the variable `i` and taint metadata propagation code for it can be eliminated from the loop. Our analysis aims to reason about such properties and can infer that `i` is not updated as a side-effect of the call.

Third, there are other implicit features introduced by the compilation process. For example, the function `get_min` deallocates the return structure pointer passed(at instruction Y in `main`) as a parameter before exiting, thereby leaving the stack pointer `ESP` unrestored after its execution. Again, this is only one instance of how stack memory is managed between activations of functions, and there are several other complications. Our analysis is able to monitor the effect of execution on the stack pointer, and therefore can reason about stack usage in an inter-procedural manner.

Finally, consider a generalization of the above case - identifying which variables (or registers) are left unchanged in the execution of the function. It is not easy to reason from the binary that the compiler generates code for function `get_min` such that even though `EBP` is modified in the function, it is left restored at the exit from the function. This requires reasoning not only about registers and memory, but also about aliasing, stack conventions and parameter passing. Our technique can, in general, identify which operands at any two points in the program, definitely contain the same value. It uses symbolic values to achieve this. Furthermore, our analysis is powerful enough to deal with capturing arithmetic on symbolic values, allowing us to extend the design to compute simple arithmetic expressions using symbolic values.

Our goal is to reason about variables in the function local scope using a static analysis. We describe the basic technique in section 3.1.2. We further show that this biased model of accurately tracking local variables references enables us to reason about higher level properties that complement alias analysis techniques developed on source code. In particular, we show that it is possible to reason about indirect references by asserting that a large set of local scalar variables can never be involved in computed memory accesses in section 3.1.3. Of course, our assumptions are made for benign programs. These assumptions may be violated by untrusted code or even in presence of memory errors. If the application of such analysis is for security and sound optimizations, additional runtime checks are needed.

Our analysis is modular and proceeds in a passes, with each pass comprising of a per-function analysis and *summarization* of the properties of each function. These summaries are used at call sites to refine the summary of the caller functions. Therefore, our analysis is scalable and can naturally handle external code or functions called indirectly through function pointers for which the called code may be unavailable or statically indeterminable.

3.1.2 Stack Variable Analysis

At source code level, there is a well-formed notion of program variables and associated scope. However, during the compilation process much of this information is lost. Compilers typically allocate stack local temporaries and registers to hold the same program variable at different points in the program, making it difficult to recognize original program variables at the binary level.

Absence of type information associated with program objects makes it impossible to determine the sizes of some program objects such as arrays, and to distinguish between scalar variables, record fields and temporary variable accesses. Although it is possible to access some of this information by referring to the relocation and debugging information, our approach does not rely on this. Instead, we perform a static analysis referred to as the *stack variable analysis*, which allows us to infer some notion of local variables.

Stack analysis uses a simple abstraction for modelling function local data objects. It uses a set of abstract data objects called *abstract variables* or *a-vars*. The idea behind the *a-vars* abstraction is that accesses to local variables and parameters in the programs written in higher level languages appear as either static stack-frame offsets or register accesses. For example, a compiler typically allocate fixed stack locations or registers in the activation record of a function, to a functions local variables statically. Consequently, we define an *a-vars* to be the set of contiguous locations between two such consecutive stack offsets, or it may be a register.

A function activation record may have several instances at runtime. Therefore, to model the dependence between stack memory objects, we can not use their static addresses. In our abstraction of the memory, we associate an *function variable store* or *FVS* with each function, which is the set of all a-vars that are accessed in the scope of the function. The FVS of each function can be thought of as a set of a-vars that are used or defined in the body of the function. Our model does not associate any a-vars with global or heap memory. In order to recover information about a-vars, we need to reason about values stored in registers as well as memory.

We recover the information about a-vars by using abstract interpretation on a abstract domain, that constitutes symbolic values as well as integer domains. By modelling the value of the stack pointer register `ESP` at the entry of a function as a symbolic constant *"BaseSP"*(base of the activation record), we can track integer-valued and stack-pointer based addresses uniformly in instructions. This is important because as can be seen from figure 3.2, compilers typically use simple integer arithmetic on the stack addresses and `ESP` for allocating stack space for local variables and accessing them, as well as for pointer arithmetic for aggregate structure accesses. The ability to abstractly model memory local to a function further enables other analyzes to explicitly track and reason about function local data objects from other memory objects. Hence, transformation and analysis built on top of this framework is not forced to be imprecise due to conservative assumptions, when reasoning about data dependencies in the presence of memory aliasing.

Formally, we define our abstract domain be elements of a lattice described in figure3.3. The intuitive semantics of each of these domain values is as follows:

- A *LocalAddressRange* denoted by $BaseSP + [k_1, k_2]$, where k_1, k_2 are integers $\in [-\infty, +\infty]$, and $BaseSP$ is a symbolic value for the value of ESP register at runtime for a given activation of the analyzed function. This value specifies the address range of consecutive bytes at offset k_1 through k_2 from the base of activation record, i.e the value of the ESP at the point of function entry.

- A *SymValRange* denoted by $X' + [k_1, k_2]$, where k_1, k_2 are integers $\in [-\infty, +\infty]$, and X' is a symbolic value. Intuitively, this captures simple arithmetic operation on symbolic values using constant, which may be useful in applications that recover aggregate field accesses. Symbolic values are introduced whenever we statically do not know the value of a variable, and lets us reason about the relation between the values of two variables.

- A set *LocalAddress (F)*, which indicates the set of all possible addresses in the activation record of a function F.

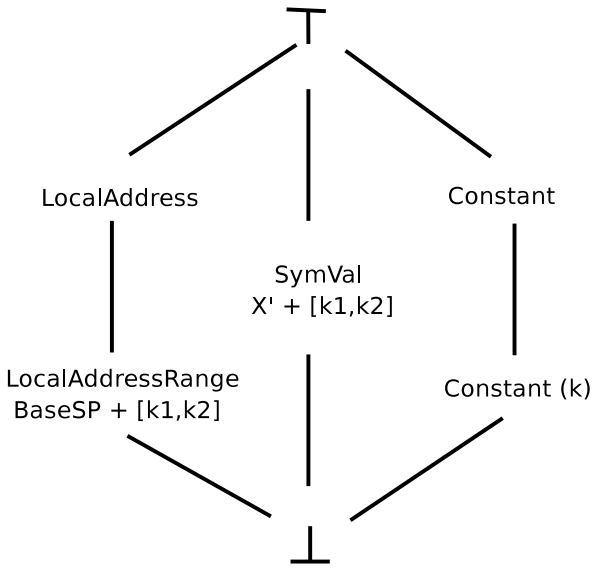

Figure 3.3: The lattice of abstract values used in stack variable analysis.

- An integer *Constant* (k), which specifies a statically known integer value $k \in [-\infty, +\infty]$. Note that we do not distinguish between non-local addresses such as addresses of global memory and constant operands.
- An integer *Constant*, specifying all integer values $\in [-\infty, +\infty]$, but not including the range corresponding to the set *LocalAddress (F)*, where "F" is the function being currently analyzed.
- A symbolic value $\top$, denoting an unknown value, including possibly all local stack addresses.
- A symbolic value $\bot$ denoting a *false*.

The basic goal of this analysis is recovering the following sets for a given function F:

- *Set1* that models the effect of execution of F on the value of the ESP. ESP may be unchanged, changed by a statically inferable constant, or may be analyzed to be an unpredictable value $\top$. Therefore, the result of this analysis is a singleton set containing the abstract value of the ESP at the point of return from F.
- *Set2* that models the maximum size of the activation record of F. This is a singleton set with a abstract value that may be a constant, or $\top$ in some rare cases.
- *Set3* is the set of a-vars accessed in F represented by their addresses.

Note that the values and sets we aim to compute depend 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 cases that called functions leave the ESP value unchanged, and access no a-vars. 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 is used to refine 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 for programs that use external libraries the code for which is unavailable. These summaries can

be generated before hand for externally called function, using similar analysis or using information present at the compilation stage. In case that is not possible, worst case summarizations can be assumed and the analysis remains sound.

Within each pass of the analysis, we aim to over-approximate the set of actual runtime values with abstract values that an a-var can hold at each program point. For this purpose, each a-var M is mapped to an initial symbolic value M', treated identical to abstract value *Constant*, at the start of the abstract execution of a function. With these initial conditions, we perform an abstract interpretation using abstract operations similar to those defined in [8] corresponding to each of the machine instructions encountered. Unlike [8], we do not model the stride information and use no affine-relation analysis since our goals do not need this. Typically, there is a small subset of the x86 instructions that we have to deal with because addresses values are not involved with most instructions. The result of other operations can conservatively result in value $\top$. 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[10] at least once in each cycle. In particular, when performing widening, the operation for widening *LocalAddressRange* value approximations, confines the widened range of addresses to limit the range of offsets from $BaseSP$ to the either the whole activation record of the caller function(denoted by $BaseSP + [0, \infty]$, or to the activation record of the callee (denoted by $BaseSP + [-\infty, 0]$), based on the initial value in the approximation. This leads to some imprecision when dealing with arrays, and can be further improved using techniques in [8] at the expense of some complexity. During this analysis, Set2 can be computed by keeping track of the minimum negative offset for ESP at any point.

With this information, there are additional sets that we can compute, for a given function F :

- *Set4* The set of initial symbolic values for a-vars, that are possibly used before being clobbered in F.
- *Set5* The set of a-vars the values of which remain unchanged as an overall effect of executing F in any context.

Using the results of the previous sets, computation of these sets can done in a similar multi-pass iterative fashion, refining the sets at each step. For Set4, we start with the initial safe approximation consisting of the entire Set3. In subsequent interpretation passes, we subtract the set of definitely defined a-vars at each instruction, and use the standard union operation at join points. Set5 can be computed after the abstract interpretation passes are complete. It will be the set of a-vars X that contain their initial symbolic values X' after the execution of all final return points in F.

The analysis assumes that programs follows a "standard" compilation model, i.e the stack grows downwards, and a standard call activation results in pushing the actual parameters on the stack, followed by the return address, and this is followed by the callee's function activation. Also, our analysis verifies that *EBP* left unchanged across a call activation, whereas *ESP* is either left unchanged or modified by a statically inferable constant. Wherever this is not possible, it optimistically assumes that they are unchanged, placing appropriate runtime checks before the next use of these registers. Although, typical OS dependent x86 ABI defines other registers, called *caller-saved registers* to be left unchanged across function calls, our analysis makes no assumption about these. There is two more implicit assumptions in our analysis :

1. All addresses of stack local memory are explicitly generated by the function and passed to other functions through memory or registers, whenever necessary. The effect of calling a function that takes the address of its parameter in memory and uses can be equivalently viewed as this case, and such cases are detected and computed in our function summaries.

2. Stack based variables are used only during the lifetime of the function activation they belong to, i.e deallocated stack memory is unused after the function exits.

Our assumptions are unrestrictive in practice, because they follow from the basic understanding of how compilers generate code, or programmers hand-write assembly. Assumption 1 basically follows from the fact that portable code for each function is generated assuming nothing about the internal layouts of the other functions active on the call stack. Addresses of stack objects are typically passed by copy, rather than inferred by arbitrary means using the value of the stack pointer register and runtime properties of the activation stack. Assumption 2 is typically the result of bugs in the program - indeed, our analysis detects most of these cases and issues static warnings about such illegal uses of deallocated memory.

3.1.3 Alias Analysis

Any analysis that reasons about values of memory based objects has to deal with the possible effects of aliasing, in order to be sound. The problem is even more acute in x86 architecture binaries, where direct memory accesses are typically fewer than 20%. Without any analysis, most indirect memory updates could update *any* memory location. Based on our stack analysis, we observed that most accesses through base pointer (ESP) and frame pointer (EBP) are statically resolvable to a single a-var. Moreover, we show that a coarse alias analysis based on the *stack analysis*, that can further *separate a-vars that can never be accessed in indirect memory accesses*, based on the assumptions about a "standard" compilation model earlier.

In real-world programs, there several problems that complicate this analysis on binaries. First, sizes of original program data objects is unknown. For example, any accesses to elements of arrays on stack appear to access all memory in the activation record. Similarly, as is typical in implementations of C++ objects, the base address of aggregates could be passed as parameters to other functions, which can be used to access any data in the activation record by using offsets that are statically unknown in the caller. In general, once the address of a variable *escapes* the scope of the function, potentially all the memory in the activation record is accessible. Second, due to undecidability of pointer analysis, call graphs are incomplete in the presence of indirect calls and due possibility of exceptional control flows such as in common implementations of "setjmp/longjmp" and signals. Third, several language features such as "alloca" and variable argument functions that access statically indeterminable size of memory in the callers, make it impossible to completely reason about a-vars in all functions.

Therefore, we limit our goal to finding the set of all *unsafe* functions, i.e functions that possibly have any of its a-vars accessible in indirect memory references. Again, we leverage on static analysis to do this. We classify any function which satisfies any of the following properties as *unsafe* :

1. Performs an indirect call which could possibly escape local addresses, i.e at the point of the call, are any addresses of local variables are stored in local memory or registers.

2. Calls a variable argument function that may access addresses of local variables. Again, we use information about whether addresses of local variables are stored in local memory range used as parameters or registers, for this purpose.

3. Has potential arrays on stack, or modifies `ESP` such that its value becomes unknown - such as the case of "alloca".

4. Passes the address of any of its local a-vars as a parameter to a function, or potentially stores it to non-local memory, or returns it as a return value.

5. Calls a function that "escapes" the address of any of the passed actual parameters.

Property 1 requires us to know the state of all local memory and whether it contains any addresses of local variables. Properties 2, and 5 rely on the results of stack analysis to get information about the a-vars that correspond to in and out parameters and registers. Property 3 is a simple check based on the results of the stack analysis - if the over-approximated set of abstract values indicates that at any point, registers involved in indirect references refer to the address ranges contain $< BaseSP + \infty >$ or $< BaseSP - \infty >$, the access is to potential stack array. Function that use "alloca" allocate a statically unknown amount from the SP, which either results in the SP pointing to the range $< BaseSP + k, BaseSP - \infty >$ or having the abstract value $\top$.

All other functions are considered *safe*. We assume that statically indeterminable indirect memory references can never refer to the local a-vars of a safe function. As a result, our analysis infers that all functions in the C program shown of Figure 3.1 are "safe", i.e no local variable of any function can be accessed in the indirect memory references.

3.1.4 Binary Instrumentation Framework

In order to deal with practical systems, we developed a fine-grained instruction-level instrumentation framework that is applicable to executables as well as shared libraries that use the ELF format on Linux. Our design is robust in the face of typical compiler optimizations such as frame pointer omission and tail calls, compiler idiomatic ways of generating PC-relative accesses, typical hand-written assembly. It also deals with the common implementations of UNIX signals as well as C++ exceptions.

The first step in our analysis is that of disassembling a binary using well-known techniques [32]. During disassembly, we also construct the control flow graph for each function, and record all its entry points. Due to limitations of existing static disassembly techniques mentioned earlier, this step requires information in the binary that identifies the start of each function. This information is normally present in executables as well as shared libraries, unless they are explicitly "stripped." Since other steps in our analysis and instrumentation don't require any symbol information, we can eliminate this limitation when static disassembly techniques are improved.

The second step performs the actual instrumentation, introducing code for taint-tracking and other security checks. This instrumentation typically introduces one or more additional instructions for each instruction in the original binary. As a result, function bodies expand, requiring them to be relocated. Rather than overwriting the original code, which can cause problems in the presence of indirect calls and jumps[1], we store the instrumented version in a separate code region. Jumps are introduced from the original function entry points to the corresponding entry points in the instrumented version[2] 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 promptly identified and fixed, and (b) attempts to evade security checks by executing uninstrumented code version will be caught.

Maintaining Application Correctness

[1] An indirect call to a function f would jump to the beginning address of f in the original code, which may fall in the middle of another function if we we replaced the original code with the instrumented version.

[2] A slight complication in this regard is that there may not be enough space available for writing a 5-byte jump instruction — this may be because of a function entry point that is very close to its end. In such a case, we search for 5 bytes of space in a region that is within 128 bytes of the original address, insert the jump at that place, and introduce a 2-byte jump instruction to jump to this instruction. In the worst case, we can use a 1-byte code, using a software interrupt as in the case of [22].

```
100:  call 105
105:  pop esi
106:  add 0x200, esi
```

```
200:  call 300
205:  add 0x200, ebx
...
300:  mov [esp], ebx
302:  ret
```

Figure 3.4: Two common sequences in PIC code.

To ensure that the instrumentation does not change the semantics of the program, we must make the instrumentation transparent to the application. There are many corner cases that need to be handled in order to deal with real applications on Linux. Instrumentation that is necessary for basic program functioning must not make any unnecessary assumptions about memory usage such as stack convention. Note that our instrumentation weaves logic for a new execution thread - such as the *taint program*, into the original. The basic philosophy in the design is to keep things simple – avoid any changes to the original program, and use a logically separate address space for the instrumentation. Our application to taint based sand-boxing uses separate tagmap addresses from the application, uses thread-specific storage wherever necessary, uses specialized I/O routines, manages all its internal memory allocation on its own, and ensures that all shared resources such as C library code are used in a re-entrant way. These are similar to the design principles outlined in [11].

Since our framework provides a bare-bones support and services for instrumentation, the most important challenge it faces is maintaining address space transparency. The approach primarily relocates code blocks leaving all data in place, we must make the change of code addresses transparent. One challenge is posed by code that computes data addresses or targets of (indirect) control-transfer from the address of an instruction. Such code is commonly used for computing the addresses of static data objects in the position-independent code (PIC) found in shared libraries. Different compilers may use different techniques in this context, with two of the common code sequences shown in Figure 3.4. Other possible uses of instruction addresses include pointers computed using PC relative offsets, and the use of PC value to compute the address of exception handling code with certain implementations of C++ exceptions via exception tables. The correctness of such code relies on the relative distances between code objects (and distances between code and data objects), and hence it needs to be modified after to ensure its correct operation after instrumentation.

The basic problem is as follows - a code address should be used in control transfer instructions, but may occasionally be used in direct data or indirect data references. The strategy to use is to identify all uses of code addresses and for data references the values should be identical to the original program. For control transfer instructions such as `return`, the translated addresses need to be used to transfer control to translated code.

Early works [32] have used compiler-specific (usually version-specific) instruction sequence matching to deal with such code. A comprehensive design philosophy is developed for dynamic translation systems such as [18, 20] by leaving all return addresses unchanged as if the program is untransformed, and using a table-lookup at runtime to translate the code addresses at runtime for control transfer instructions. Our design tries to achieve the same in a different way. We aim to identify all sequences that access return addresses (code addresses, more generally) in a uniform way using a static analysis described later to achieve fair robustness. Specifically, we analyze the callee code to check if it uses the return address for any purpose other than returning. For each such use, e.g., a `mov` of a return address to register R, we introduce an instruction that adds a constant k to R, where k is the difference between the original location of the call instruction and

its new location in the instrumented code. In effect, this intercepts all uses of code addresses in non-control transfer instructions and fixes them up at runtime. Practically, the approach used in dynamic translation systems seems more robust, as our approach makes an optimistic assumption that all references to return addresses are statically identifiable. We may miss some cases when return address may be accessed using techniques that our static analysis cannot precisely reason about such as indirect memory references using global pointers, although our technique has successfully handled the common cases we have encountered in our limited experience. Our suggested approach does have important benefits during incremental development of the tool. In the alternative technique, the application would not run until all control transfers in all libraries are transformed. Detecting the smaller set of accesses that use code addresses in non-control transfer operations in a uniform way deals with almost all cases. This is beneficial to proceed with development; once the instrumentation is stable enough we can switch to the alternative technique for even more complete robustness.

Chapter 4

Optimization Techniques For Binary Taint Tracking

Performance is a critical factor for deploying proactive defense techniques on end host applications. End users will typically turn off security mechanisms as "nuisance", if it interferes with their business. This is certainly true for most applications we target : if we slow down plugins used for viewing real-time streaming media or for displaying documents downloaded by the user, it will be glaringly visible and will impact the usability of the system. Therefore, we focus on optimizations to reduce the overhead of our technique. The primary overhead in our approach is due to fine-grained instrumentation for taint tracking, and is the primary subject for optimization.

Fine-grained dynamic information flow tracking has become a very powerful tool in computer security, having found extensive use in detecting a wide range of attacks, malware analysis, and so on. However, previously reported works on binary taint tracking have incurred large overheads - 37 times slowdown in [23], with recent improvements to an overhead of 3.6 times in [25]. Most of these techniques have used dynamic code instrumentation for reasons of robustness. Dynamic translation systems[18, 20, 9] typically incur large overheads because of two main reasons :

- Overhead of code analysis and instrumentation is incurred at runtime. This indirectly affects the choice of optimizations, since expensive analysis required for them are avoided.
- Dynamic methods can not perform perform sound static analysis in the absence of complete flow-graphs, thereby limiting the kinds of analysis applicable.

Moreover, there is significant overhead incurred whenever the dynamic translation framework discovers a new block of code that it has to dynamically transform. This is typically not reported in research experiments because tests are run for a long time and the average values are not indicative of the initial startup overhead. However, from a practical standpoint this is a significant drawback for usability in interactive applications such as viewing documents online, as users will not wait for minutes for the plugin and the host to respond.

We address these fundamental limitations by shifting the analysis and instrumentation burden offline without sacrifing robustness. Using our static binary rewriter BinStat, we transform binary versions of several large applications such as Firefox and Konqueror(default browser on KDE). Further, we develop techniques to lower performance for taint tracking below 90%, which bests the contemporary techniques by a factor of 4. At the same time, we maintain the soundness of taint tracking, i.e to never miss propagation of taint metadata for data, and do not neglect any avenues of attacks outlined earlier. Our technique are applicable to real multi-threaded applications on Linux/x86 and requires no additional hardware features such as dedicated registers as assumed in

[25]. We describe these *general optimizations* that can be used in binary taint-tracking in section 4.2. These optimizations are used to implement our confinement of shared memory extensions.

Information flow techniques have much wider application than our application such as cyber attack detection, signature generation and enforcement of OS integrity. There are several other optimization techniques that may be applicable in a more traditional setting of taint propagation on benign code where attacks due to concurrency inherent in multi-threaded programs is not a practical issue. Under these assumptions, a much more sophisticated set of optimizations become possible using the techniques described in Chapter 3, that recover sufficient higher level program structure. These are subsequently described as *higher-level optimizations* in Section4.3. Some of these have wider applicability to other metadata propagation approaches such as memory error checking, as well as to techniques applicable on source code.

4.1 Basic Transformation

Our primary focus in this thesis is to perform general taint tracking on binary code which has been used extensively in security applications. In particular, our technique associates one byte of taint data with each four byte word in memory. Sharing of taint data between adjescent bytes of a word can lose precision when dealing with character arrays, but does not sacrifice the ability to detect attacks.

Taint propagation requires an initial marking phase, that defines the interface for untrusted input. This is usually a set of I/O interface functions such as file or network API. Data received from untrusted sources such program inputs from network or I/O reads, is marked potentially "unsafe" by setting the associated taint data to 1. In other applications such as our approach of confining extensions, this data may correspond to all data originating from the untrusted plugin. As this data is manipulated by the program, the corresponding taint bits of operands are updated. More specifically, the application is instrumentated in its binary form to perform taint propagation, i.e, marking the results of any logical, arithmetic or assignment instruction as tainted if and only if any of its input operands is tainted. The form of taint tracking described here is largely same as what is performed on the host application, with some policy changes with regards to making it secure in presence of untrusted extensions. These changes are minor including reversing of taint and data accesses and some operations for accessing misaligned accesses. We eliminate these in the discussion below and focus on the crux of the instrumentation technique that is generally been used in taint tracking works[31]. Similar to most previous works, we do not track control dependence, as this is sufficient to address a wide range of attacks and at the same time has low false positives.

For general information flow tracking, in addition to taint propagation security checks have to be added as governed by an application specific policy. For example, to detect all code-injection attacks the policy has to enable checking of each computed control transfer. As another completely different application in sandboxing, the checks have to ensure that the argument data passed to a security sensitive operation such as execve system call is never tainted.

Taint information for memory is stored in a separate byte array referred to as the *tagmap*. Taint for registers can be stored in a special area that is specific to each thread. On the Linux platform, thread specific data is stored in a separate data segment accessible by using `%gs:0x0`[11]. The program transformation is a simple rule driven procedure, that takes an original program instruction as input and outputs a corresponding *taint operation* based on an action table indexed by an opcode. Taint operations use the tagmap, register taint data and the address of program memory operands for performing the taint propagation. For all further optimizations, we refer to the set of taint operations as the equivalent *taint program*. Program execution interleaves between

```
                        movd eax, xmm0          movd eax, xmm0          movd eax, xmm0
                        movd xmm1, eax          movd xmm1, eax
                        movd esi, xmm1          movd esi, xmm1
                        test 0xa, al            test 0xa, al
                        jz L1                   jz L1
                        or 0x8, al              or 0x8, al
add ebx, edx            L1:  add ebx, edx       L1:  add ebx, edx       add ebx, edx
cmp ebx, 0              cmp ebx, 0              cmp ebx, 0              cmp ebx, 0
                        lahf
                        test 0x42, al           test 0x42, al
                        lea [ebp+0x1c], esi
                        shr 2, esi
                        setnz [tagmap+esi]      setnz [ebp+0x801c]      movb 0, [ebp+0x801c]
                        sahf
mov ebx, [ebp+0x1c]     mov ebx, [ebp+0x1c]     mov ebx, [ebp+0x1c]     mov ebx, [ebp+0x1c]
                        movd xmm1, esi          movd xmm1, esi
                        movd eax, xmm1          movd eax, xmm1
                        movd xmm0, eax          movd xmm0, eax          movd xmm0, eax
je 0x40000              je 0x40000              je 0x40000              je 0x40000
```

Figure 4.1: Figure shows (a) Untransformed code snippet. (b) Instrumented code after flag-liveness optimization. (c) Instrumented code after taint-stack optimization. (d)*Fastpath* version of instrumented code.

executing original program instructions and instructions of the taint program, and therefore a *context switch* is performed between the two logical threads of execution. It is useful to think of all the state required for performing the taint operation as well as the context switch to be stored in *virtual taint registers*. We employ some optimizatation techniques described in [25], to reduce this overhead such as by using cheaper `lahf/sahf` instructions, and performing a flag register liveness analysis across the function. An additional cost in our system is that of saving and restoring physical registers used virtual taint registers, since we assume no availability of dedicated registers for instrumentation, based on the practical observation that code for production use is unlikely to have any general purpose register unused. Instead we use static analysis based methods to perform the register re-allocation, described later in the chapter. The tagmap array can be allocated statically in memory or by using an on-demand allocation as in [31].

4.2 General Optimizations

Our basic instrumentation required an expensive *context switch* from application code to the taint monitor at each instruction, requiring save/restore for physical registers needed to perform taint-related operations. As can be seen in Figure 2.1, this typically adds 10 to 20 instrumentation instructions for every instruction in the original program. Worse, about 10 additional memory references are added for each original memory reference, and has a base overhead of about 10 times slowdown. We describe five optimizations we implement based on our sound static analysis. The first three of these optimizations are generally applicable to both trusted and untrusted code, while the others are safe only in the context of trusted code.

- **Reducing Monitor State for Fast Context Switch**. As a first optimization, we optimized the saving and restoring of registers in two ways. First, we reduced the number of registers

required by our taint monitor. Initially we required 4 physical registers — 3 for realizing the virtual registers in Figure 2.1 and 1 for a pointer to the thread-specific register taint data. We identified an interesting way to eliminate one of these registers by using the CPU flags to perform taint computation. Second, we packed the taint for all CPU registers into a 8-bit quantity, i.e 1 bit of taint per register on the x86. This 8-bits plus 8-bits needed for saving flags could share a single 32-bit register, thus reducing the total register requirement to 2 from the original 4. The effect of these low-level optimizations can be seen in Figure 4.2, which also shows the effects of several additional optimizations described below. The technique of storing the taint bits of all registers within a single CPU register allows more efficient instrumentation: the taint of multiple registers can be checked using a single instruction.

As a second optimization, we further reduced the number of memory operations by utilizing certain architectural features of Pentium. Specifically, we employed XMM registers (which are rarely used is general-purpose code) as a backup for registers needed for instrumentation, instead of saving them to memory. These registers are not used by compilers since they are much slower than CPU registers — they provide access times comparable to that of L1 cache, i.e., their access times are comparable to memory. However, they provide an important benefit in the context of thread-specific instrumentation, as they are private for each thread.

- **Register use optimization**. The goal of this optimization is to reduce the context switch overhead by improving the selection of physical registers that are used as virtual registers. We divide each application code basic block into sub-blocks such that each sub-block has at least two unused registers. These registers can be used for taint computation. Their values needs to be saved only once per sub-block instead of once every instruction. Other more complicated ways of performing optimizations such as performing new register assignment after rewriting the whole program may yield slightly better performance. However, our design leaves the original instruction's memory and register usage untouched. Therefore, the design is simpler, aids debugging during development and may later aid automated reasoning about correctness – in this regard, separating taint program code from the original has significant gain.

- **Flag liveness optimization**. Our basic instrumentation saves flag before each snippet of instrumentation code and restores them afterwards. This is wasteful since flag values aren't live in most parts of the original code. Typically, fflags are set by an arithmetic instruction (e.g., `cmp`) just before a conditional branch, which uses the flag values. Based on this observation, we implemented a sound static analysis for flag liveness, and eliminated the `sahf`/`lahf` instructions when they are not live.

- **Use of dedicated taint stack**. We split the tag space into two regions: one containing taint data for stack, and the other (i.e., the global tagmap array) for all other memory. Actually, there is one taint-stack for each thread stack. The taint stack is located at a fixed offset from the main stack, and holds the taint values corresponding to the stack data. (For simplicity of illustration, this offset is assumed to be `0x8000` in Figure 4.2.) Use of taint stack avoids the need for address computation for local variables, as well as bit-shift computations. This can be a significant gain for most code, since local variables often account for the vast majority of memory accesses made by most programs [31].

  If an access can be statically determined to be within the stack, then the corresponding taint updates are directed to the taint stack, or else they go to the global tagmap. However, for indirect accesses, a two-step operation is needed: to obtain the taint for an address a, $tagmap[a]$ is first accessed. One of the bits in $tagmap[a]$ indicates if a is on the stack, and

if so, the taint update is directed to the stack, i.e., to the location $a + X$ where X if the offset between stacks and taint stacks. (Note that by assuming a constant value for X, we restrict all stacks to have size less than X.) This two-step process increases the costs of taint update of indirect memory references, so the optimization may not work well for some programs that perform far more static memory accesses as compared to local variables. We point out that this two-step process is necessary for ensuring *soundness of taint data* in the case where arbitrary memory corruption can happen between any two instructions, such as the one addressed in that concurrency attacks outlined in previous chapters.

Local variables and parameters are accessed using constant offsets from `EBP` or `ESP`. We can direct the taint updates for such accessed to the taint stack *only if* EBP (and/or ESP) are known to point to the stack region. We verify this at the beginning of each function, and perform a static analysis to verify if this condition will continue to hold throughout the execution of the function, using static analysis techniques in BinStat. This analysis succeeds on about 75% of the functions we have analyzed, but for the rest, it fails due to our conservative modelling of indirect calls and setjmp/longjmp, and several features such as alloca, hand written assembly, callee functions that de-allocate parameters before return, frame-pointer elimination optimization, and involvement of ESP in arithmetic instructions (other than those used to align the stack to alignment boundaries). For those 25% of functions, EBP- (or ESP-) relative accesses are handled in the same way described above for indirect memory references. The same is done for all stack accesses made by untrusted code since it is difficult to perform sound static analysis on untrusted code — such code may violate many assumptions made by such analysis.

- **Generating multiple code versions**. The intuition behind this optimization is that for a large part of the program execution, the host system has very little memory interaction with the untrusted code and therefore spends considerable time propagating "safe" data from one memory to another. 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 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 memory taint check and transfer of control to *slowpath* version, if necessary. Figure 4.2 shows the fastpath code. Figure 4.2 shows the slow path code, with one change: instructions to check if register taint is zero are added to the end of each basic block, causing a transfer to the fastpath version in that case.

  Although conceptually similar to the fastpath optimization developed in [25], our fastpath optimization is more advanced due to the use of better static analysis. For instance, they have to perform runtime checks at the beginning of each code block if the registers are tainted, whereas we infer this by static analysis. Second, since their fastpath performs no taint update operations, it can be used only if all memory locations that are written in the block are untainted at the start of the code block.

4.3 Higher-level Optimizations

Static instrumentation systems have a critical advantage over the dynamic instrumentation systems - the ability to analyze the complete flow graph of a function, thereby enabling reasoning about

stronger properties at each instruction. Also, the analysis and instrumentation overhead is shifted offline, which affords the possibility of much deeper analysis that yields results closer to optimal. We perform three basic optimizations considering the semantics of taint program. Previous approaches [23, 25] have not fully utilized instrumentation specific semantics, and therefore have not been able to apply optimizations analogous to those applied in compilers for the taint program semantics.

4.3.1 Tag Sharing

The general intuition behind this optimization is that it possible to perform static taint tracking on the taint program, falling back to dynamic updates only when there is static indeterminable ambiguity in taint values. During the static taint analysis on the program, we introduce symbolic taint variables whenever we load statically unknown taint values. These taint variables can be logically shared by multiple program variables at a given program point. This eliminates the need to propagate taint for variables within the same tag variable.

Most previous works in taint tracking associate a *storage view* for associating taint metadata with data, i.e access to a metadata for data X, is $f(X)$, where f is a function to compute the address for metadata stored for X such as an array lookup operation. The function f is invariant across the whole program, which results in certain inefficiency in accessing taint metadata. This is typically because compiled code uses several temporary memory and registers imposed by architectural constraints, such as limiting at most one memory operand for most instructions on the Intel 32-bit x86. Consider the common case of a program data which has a long live range. On a register starved architecture such as 32-bit x86, it is stored in function local temporaries being moved in and out of registers when used. Associating taint metadata data using a storage view, the taint operations would mirror the original program, copying taint metadata for these temporaries multiple times. Logically, we can eliminate many of these taint operations, if we can statically infer that each program point, the metadata access is located in an already allocated taint variable shared by other variables.

Based on our ability to statically analyze the whole program flow graph for a function, we perform standard analyses on the taint program. We first represent the program in SSA form, giving a new tag variable whenever we can not precisely identify which tag variable is accessed using our static analysis and at ϕ nodes. Then, treating taint for constants as "0", we can perform analysis similar to constant propagation. Using this, it is possible to recover a certain set of variables that require no dynamic taint tracking - such variable `i` in `main` of Figure 3.1 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 3.2, just before instruction at `.L11` in function `main`, `esi, edi and eax` share the tag variable. Further, in the loop with the help of our analysis we can ascertain two facts -

- Registers `esi and edi` are left unchanged by the execution of `get_min`.
- They are only involved in pointer arithemetic and comparision operations, both of which do not change the taint associated with them.

As a result, `esi and edi` can share the tag variable throughout the execution of the loop body.

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 3.2. All of this has an additional impact on register pressure and cache performance incurred by the taint program.

In order to enable the tag sharing optimization, we must not sacrifice soundness of the taint tracking, i.e we should never miss marking the "unsafe" data with "unsafe" taint. For this reason, we must treat memory locations that could be accessed in indirect memory references and as side-effects of executing other functions conservatively. For non-local variables, we associate their taint using the standard storage view, i.e by associating a corresponding position in the tagmap array. For local variables we rely on our stack analysis and alias analysis to decide which function local variables are suitable for this optimizations. Stack based variables of functions that are checked to be "safe functions", are optimized, because their references are identified statically and they can never be accessed in indirect memory accesses. Even for "unsafe" functions that are marked only because they call indirect functions or variable argument functions, they could later be selectively treated suitable for this optimization, with the additional care that virtual taint registers used for storing taint variables are flushed before an unsafe function call to their tagmap locations.

4.3.2 Virtual Taint Caches

Our ability to selectively distinguish stack-based accesses from other accesses, enables another optimization. Similar to our use of `al` in Figure 4.2 to hold taint data for registers, we can now use virtual taint caches for virtual registers used in the taint program. Specifically, for most local stack variables that we recover in our static analysis, we can associate fast taint access for them. These caches are implemented as thread-specific general purpose registers.

In order to perform this optimization, we first represent the taint program in a static single assignment or SSA form. Then, we build an interference graph for all taint program variables. We perform a greedy graph-coloring algorithm to decide if the number of bits required to allocate register taint bits to all taint variables is lesser than a constant k, (fixed to be 54 in our implementation currently). If so, we perform a graph-coloring based register allocation for taint data. Using techniques described in section 4.2, we find free registers and use those for allocation the virtual taint cache, just as done for virtual taint registers .

As with the tag sharing optimization, we must be sure that cached taint data is in sync with the taint tagmap data, whenever a local stack variable can be accessed in an indirect reference, or as a side-effect of a call. This is referred to as a taint cache flush. To do so, we identify points in the programs where the taint for a stack based variable must reside in the tagmap to avoid unsoundness due to associating two locations with taint data. Such points could be function call locations or indirect memory accesses. For most stack data of nearly 75% of the functions which are "safe" cache flushing is unnecessary, since they can never be involved in indirect memory references. This achieves significant speedup in taint access for a large part of the program execution, leading to high performance gains.

Whenever possible memory errors becomes a concern in the application, caching can lead to some unsoundness that should be eliminated. Essentially, this is because there are two physical locations for the same taint data - one in the tagmap and other in the cache. In our approach, this can be handled as follows. We make all stack variables for "safe" functions `inaccessible` by using one bit called the permission bit in the tag map. Since only a few functions are found to be unsafe, we can explicitly set their permission bits to accessibile or "1" at entry, as only its local variables are supposed to be accessibile indirectly. We check this permission bit before any indirect memory access in the program. Any memory errors that access cached local tag variables will be caught at runtime as the permission checking will fail.

4.3.3 FastPath for Local Variables

This optimization a simple extension for the fastpath technique described earlier. It combines the benefits of fastpath with those of storing taint data in caches which are implemented as bits in general purpose registers.

We generate two versions of code as outlined earlier - a fastpath version and a slowpath one. The fastpath version assumes that all registers and stack based local variables have "safe" or "0" taint. As a result, further checking of taint at memory loads from stack resident memory, and clearing of taint data for local variables is avoided. As a result, for the code in Figure 4.2, we eliminate all taint tracking and the code looks same as the uninstrumented version.

The control switches from fastpath to slowpath only upon checks on non-stack memory loads. Switching back from slowpath to fastpath is fast, as it requires checking of all virtual caches which are largely based on registers. For functions that may have taint data for local stack variables in the tagmap (or taint stack), this check may be performed at a coarser granularity such as at start of "cold blocks" or the optimization may be eliminated all together for "unsafe" functions (as done in current implementation).

Chapter 5

Implementation

BinStat is built with modifications to [32], with some changes to its disassembly engine. It uses a library called udis86 for performing all instruction level functions. For instrumentation, there is a string based interface to a modified version of nasm assembler, which acts as instruction assembler. The total size of all the code in the BinStat framework, including code for taint processing and optimizations is around 90KLOCs. Most of the code is C++ based, while some parts are written in C.

For interception of system calls, the C Lib is transformed using BinStat. Specifically, only basic blocks that make system calls are transformed for interception. Transforming shared libraries such as the C and C++ libraries for taint propagation requires handling of many special cases and is not fully done yet. Additional limitations are imposed due to certain dynamic ELF relocations that the dynamic linker fixes up at runtime by updating addresses in the original program code section. To make these updates happen in transformed code, such dynamic relocation information has to be rewritten. Some of the libraries in Mozilla Firefox have these - thus have not been transformed yet. Whereas most libraries for Konqueror have been successfully transformed, libraries in Firefox pose more problems. The suspect in these cases seems to be some handling some transparency issues as pointed out in [11].

Implementation for confining untrusted code is also limited. The CFI checks and ABI semantics are currently not implemented. Also, data concurrency attacks require out-of-order data and taint access, which is also currently ignored in the implementation.

The optimizations are basically tested on small to medium size programs consisting largely of the SPEC benchmarks and some CPU intensive applications such as pdf-to-ps converters, and the like. The basic robustness of the transformation engine is well tested transforming large programs such as Firefox and Konqueror. The static analysis is implemented with a few approximations - higher level optimizations are only applied on what functions are classified as “safe”. This is overly conservative and better results may be possible once the design is fully realized. A possible unsoundness in the results given exists because of a bug of the optimistic handling of PIC calls - we currently assume that external calls made using PIC code uses no arguments.

Chapter 6

Experiments and Evaluation

6.0.4 Policy development and enforcement

The primary goal of these experiments was to show that it is relatively easy to apply well-known policies for stand-alone applications to their plug-in counterparts. In most of our experiments, we had to make no changes to the sample policies, with requirement for some adjustments to be made when browser performs certain actions on behalf of the plugin.

Our primary target in policy enforcement experiments was Konqueror, since it supports many more plugins available as shared libraries than the Linux version of Firefox.

- **Konqueror `kpdf` viewer plugin**. For the purpose of evaluation we considered *libkpdfpart.so* which has 3143 functions and pervasively uses the KDE UI libraries for it actions. We applied existing policies developed for similar document viewer, taken from [27]. The basic policy applied was to allow only file reads, and restrict file writes to files "owned" by the application (basically, its preference files). This default policy doesn't permit the extension to make any network reads and writes. We relaxed this policy so that it could write a log file for debugging purposes, since the plugin we used was compiled from source and had default debug logging enables. As an aside, this also confirmed the correctness in capturing the behavior of the plugin when calling browser functions.

  These policy restrictions were imposed on system calls that had tainted arguments, or were made with the context flag C_u set. For system calls where none of these conditions hold, no restrictions were applied. As a result, browser functionality wasn't restricted in any way by this policy.

- **Firefox with VLC media player plug-in** In this experiment we used a media player with Firefox. To this plugin we applied a policy developed in [24] for a stand-alone version of a similar player ("kmplayer"). This policy restricts the player to make network accesses to a local DNS server and to remote web sites, but disallows writes to any files not owned by it. Policies that further restrict its network access to improve security can also be applied. For example, we can restrict the plugin to make socket connections and data transmission system calls to a local DNS server, and a specific URL that the browser has explicitly provided to the plug-in — typically the URL of the streaming media file that the medial player was invoked with.

- **Apache `mod_config_log` module**. In this experiment we took the case of an extension that is less trusted than its host system, specifically, a logging extension of the Apache web server that performs a simple operation such as recording information about requests received by

Program	Untrusted	Trusted (Unopt.)	Trusted (Slow Path)	Trusted (Fast Path)
alvinn	37	613	364	123
ammp	15	254	154	64
art	15	288	129	34
compress	75	1130	350	100
equake	17	265	109	29
espresso	72	1106	380	93
go	108	874	359	100
gzip	139	874	530	101
m88ksim	60	1293	344	80
parser	123	1545	457	112
Average	66	824	318	83

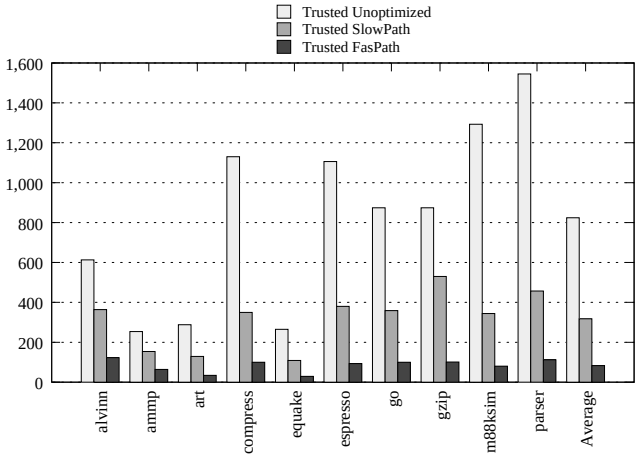

Figure 6.1: Execution overheads in percentage (%) for the general optimizations described in section 4.2.

the server into log files in a specified ("logs") directory. As a policy for such a plugin, we restrict writes of data tainted by this library to only files in the "logs" directory.

6.0.5 Effectiveness of Optimizations

For measuring performance improvements due to our optimizations, we tested our performance on 10 CPU intensive programs from the SPEC95 INT benchmarks. All programs were compiled with gcc-4.1 with typical O2 optimization levels.

6.0.6 Evaluation of General Optimizations

Our basic transformation incurs an overhead ranging from 2.5 to 15 times with an average of 8.3 times. Our low-level optimizations reduces this overhead fairly uniformly on all programs, by about 200%.

Our sub-block optimization yield a significant improvement of about $200\% - 230\%$. The reason for this can be related intuitively to our results in Table 6.2, which shows that an average number of instructions requiring taint-tracking per sub-block is 2.27. This is roughly equivalent to eliminating the need to save/restore registers atleast 55% of the instructions. This translates to eliminating roughly 6 register to XMM (or memory) transfer operations for these 55% instructions, as can be inferred from the transformation shown in Figure 4.2.

Similarly, we achieve a 140% reduction in flag save restore, since our static analysis results show that less than 10% of the instrumented instructions require flag saves after the `eflags` register liveness analysis. This reduction does not have as much of an impact as our sub-block optimization, because flag/save restores require only a `lahf/sahf` and register to register movement instructions involving `eax`, which are relatively cheaper as compared to memory accesses.

Our taint stack optimization gives us an additional improvement of varying from $5\% - 120\%$ depending upon the nature of the program, with an average of 60%. From Table 6.2, we can see that only about a 20% of the accesses are statically unresolvable. While our transformation makes indirect accesses more expensive, it optimizes the rest. However, it must be borne in mind that the static figures are not exactly representative of the way compilers generate code: minimizing the number of memory accesses in loops. This is why these reductions are not as dramatic.

Program	Unresol	Stack	Static	FlagsLive	Ins/SBlock
alvinn	19.2	75.0	5.8	8.75	1.77
ammp	31.5	61.3	7.2	7.04	1.77
art	13.1	57.1	29.8	2.7	2.30
compress	16.1	5.1	32.9	5.5	2.57
equake	21.4	57.3	21.3	7.5	2.52
espresso	30.7	62.5	8.8	4.7	2.70
go	30.9	63.4	5.7	5.8	2.18
gzip	19.5	52.3	28.2	7.3	2.27
m88ksim	16.5	67.0	16.5	4.95	2.39
parser	28.8	62.8	8.4	4.67	2.32
Average	22.7	60.9	16.4	5.89	2.27

Figure 6.2: Static analysis results for local analysis of functions: memory accesses identified to be in different categories (stack, static, or unresolved), the percentage instructions that required a flag save, and the average number of instructions requiring taint tracking in each block.

Our *fastpath* optimization performance is hard to accurately measure as it depends upon the memory interaction between the trusted and untrusted components. Therefore, we present the performance of the fastpath and slowpath codes separately. Since the transfer between the two versions requires a simple `test reg, reg` and `jmp`, the switching overhead is minimal. As can be seen, this optimization improves the performance of the host system drastically, since systems such as browsers and servers will largely operate in `fastpath` mode, involving little interaction with common extensions. Previous work[25] that optimize based on this same observation have shown that this ratio is typically 98% on server programs such as Apache. Using this as a metric, we expect overheads incurred by our approach to be below 90%.

Our performance overheads are much better than those previously reported. The fast path overheads are comparable to source-code based taint transformation techniques [31] that have yielded the best performance figures to date. As compared to binary-based techniques, the best reported results are about 3.6 times [25]. Some of this improvement is because these systems are implemented of dynamic translation frameworks that do not enable complete benefits of static analysis based aggressive optimizations.

Finally, our performance for the untrusted code is shown in Figure 6.1. The performance numbers are comparable to the fastpath version of trusted code even though taint has to be written to destination memory only in the case of store operations. This is mainly because our design incorporates thread-safe tainting, which requires taint marking to be performed before and after each memory store operation.

6.0.7 Evaluation of higher-level optimizations

Figure 6.3 shows the incremental effects of applying higher level optimizations. These are applied incrementally on the slowpath version of the code after performing the general optimizations. We have currently not implemented the common subexpression elimination on taint variables. Each SSA operand is given an associated taint variable that is given register bit storage that is updated from the tagmap dynamically. Therefore, the tag sharing optimization reduces the number of tracked variables and results in reducing the performance penalty by nearly 90% of the base execution of the uninsrumented program. This also includes elimination of tainting due to constant propagation of "0" taint that eliminates variables which always have "0" taint. Tags are stored in register caches - we utilize part of `VR1` and another virtual register `VR4` to get 56 bits of storage for taint data for local variables.

Program	Basic	Tag Sharing	Taint Liveness	Fast Path
alvinn	364	245	242	134
ammp	154	121	89	60
art	129	67	35	7
compress	350	220	150	79
equake	109	68	60	30
espresso	380	292	103	38
go	359	295	178	79
gzip	530	397	228	100
m88ksim	344	180	100	52
parser	457	328	167	99
Average	318	221	135	68

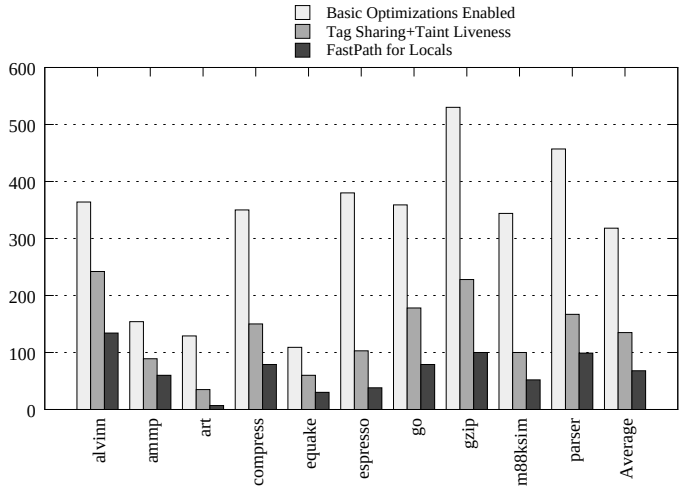

Figure 6.3: Execution overheads in percentage (%) for the high level optimizations described in section 4.3.

Further, based on placing the security checks at function entry and exits we perform taint liveness analysis, pruning away unnecessary taint updates for local variables. Of course, all taint updates for non-local and unsafe local variables are performed. The result is a further reduction to a 139% over the uninsrumented code execution.

Finally, we perform the fast path optimization on the local variables as well as registers. The dominant execution falls in the fast path code and this runs at a overhead of 67%.

6.0.8 Robustness of Binary Transformation system

To test the robustness of our basic instrumentation system we transformed various stand-alone applications with taint transformation, as well as shared libraries and we successfully tested to work properly. Apart from the experiments highlighted in this section we have transformed a variety of stand-alone applications such as gimp-2.2, gaim, pdftops, xmms, and a statically linked version of Firefox (56K functions) as well as a dynamically linked one (242 functions). These have been running on our Ubuntu system for the past few days.

In order to get useful taint information at the lowest interface, we transformed 12 KDE libraries (written in C++) such as kdelibs 3.5.6, kdebase 3.5.6, kdecore, kdegraphics, and 5 similar ones in Firfox. The binary versions of the browsers themselves are relatively small (242 functions in Firefox, and 11 in Konqueror), while most of the functionality lies in the libraries (6.6K functions in libkdecore, 8.0K functions in libkio, 7.9K functions in libkdeui).

For plugins, we experimented with several other library versions of media players such as gxine and Open helix player, limited only by the availability of binaries with enough symbol information for disassembly.

6.0.9 Defense against Attacks

To evaluate the effectiveness of our implementation, we used several synthetic attacks. For this purpose, we simply treated a logger module in Apache web server as untrusted, modifying its source to perform attacks on Apache. Specifically, we tested against:

- function pointer corruption attacks by corrupting the GOT entry for *write* library call in

`libapr-1.so`, and checked that the control context at the next system call is correctly identified as being made in the context of untrusted code.

- To detect attacks against direct corruption of data and data pointers, the logger module was programmed to corrupt certain known global buffers in the SSL module. As a result of the attack, these buffers got tainted. As a result of operations performed by SSL module subsequently, this taint propagated to a `write` system call. Since we don't expect any data written by the SSL module to be tainted, the attack gets detected at this point.
- For evaluating our defenses against ABI violations, we corrupted all callee-saved registers, and our system recovered from this attack by restoring those registers.
- To check our system call interception works as expected, we modified the source of our sample plugins to perform invalid system calls (unexpected system calls or with unexpected argument tainting), and successfully detected them.

Bibliography

[1] http://developer.mozilla.org/en/docs/Plugins.

[2] http://httpd.apache.org/docs/1.3/mod/.

[3] http://vil.nai.com/vil/Content/v_vul25979.html.

[4] Analysis of a malicious jpeg attack. www.vigilantminds.com/files/jpeg_attack_wp.pdf.

[5] A. Moser and C. Krügel and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[6] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security (CCS)*, November 2005.

[7] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine applications. Technical Report TRCS99-15, 31, 1969.

[8] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. Compiler Construction (LNCS 2985)*, pages 5–23. Springer Verlag, 2004.

[9] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-based transparent and comprehensive control-flow error detection. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 333–345, Washington, DC, USA, 2006. IEEE Computer Society.

[10] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. *Lecture Notes in Computer Science*, 735, 1993.

[11] Derek Bruening. Efficient, transparent, and comprehensive runtime code manipulation, 2004.

[12] Karl Chen and David Wagner. Large-scale analysis of format string vulnerabilities in debian linux. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, 2007.

[13] Mihai Christodorescu and Somesh Jha. Testing malware detectors. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 34–44, New York, NY, USA, 2004. ACM Press.

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

[15] 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.

[16] Ulfar Erlingsson and Fred B. Schneider. Irm enforcement of java stack inspection. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 246, Washington, DC, USA, 2000. IEEE Computer Society.

[17] H. Feng, O. Kolesnikov, P. Folga, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, May 2003.

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

[19] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, New York, NY, USA, 2003. ACM Press.

[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] Scott McPeak, George C. Necula, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for C program analysis and transformation. In *CC*, pages 213–228, 2002.

[22] 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.

[23] 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.

[24] Niels Provos. Improving host security with system call policies. In *12th USENIX Security Symposium*, 2003.

[25] 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 International Symposium on Microarchitecture*, December 2006.

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

[27] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, New York, October 2003.

[28] David A. Wagner. Janus: an approach for confinement of untrusted applications. Technical Report CSD-99-1056, 12, 1999.

[29] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1994.

[30] Yi-Min Wang, Roussi Roussev, Chad Verbowski, Aaron Johnson, Ming-Wei Wu, Yennun Huang, and Sy-Yen Kuo. Gatekeeper: Monitoring auto-start extensibility points (aseps) for spyware management. In *18th Conference on Systems Administration LISA*, pages 33–46, 2004.

[31] 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.

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