A Practical Technique for Containment of Untrusted Plug-ins

Prateek Saxena*, R. Sekar, Mithun R. Iyer and Varun Puranik, Stony Brook University, NY.

Abstract

Previous defenses against untrusted COTS software have been focused primarily on stand-alone applications. We develop a new approach in this paper that enables these defenses to be applicable to the context of shared-memory extensions (SMEs) available in the form of binaries, such as browser plug-ins that have become very popular in the last few years. Central to our approach is a new technique for *secure attribution* of sensitive operations to an SME or its host application. This enables selective confinement of an untrusted SME's actions *without having to restrict the activities of its (trusted) host application.* Our approach requires no modifications to SMEs or their host applications, and does not require source-code access. It is robust against maliciously crafted SMEs that actively attempt to evade our defenses. Our experimental evaluation shows that the approach is effective with contemporary plug-in architectures and several SMEs, and introduces relatively low overheads.

1 Introduction

Users are increasingly relying on untrusted software in their daily activities such as viewing documents and images, listening to music, watching video, instant messaging, multimedia communication, file-sharing, and playing games. The explosive increase in malware, which often hides in software from untrusted sources, highlights the need for secure execution techniques for such software.

Previous techniques for securing untrusted code have been focused mainly on stand-alone applications. *System-call based policy enforcement techniques* [34, 3, 23, 28] are based on limiting an untrusted application's system calls and their parameters. *Information-flow based integrity techniques* [5], which have experienced a resurgence of late [18, 15, 31], are based on labeling the outputs of untrusted applications with low integrity, and ensuring that low-integrity data cannot flow into (and hence influence) the operation of benign applications. Unfortunately, these techniques do not address the emerging trends towards plug-in and module-based software architectures, where the behavior of a (benign) application is extended by adding *software extensions* to it. Examples of such shared-memory extensions (SMEs) include:

- browser plug-ins for viewing various document formats, multimedia, presentations and animations,
- libraries for image decoding and display,
- audio and video codecs,
- Photoshop (or GIMP) image processing filters, and add-ons to packages such as Office, Gaim, Apache etc.

We develop an approach in this paper that enables existing untrusted code containment techniques to be applied to SMEs. Our technique is able to selectively confine the actions performed on behalf of an untrusted plug-in without restricting the actions performed by the host application. Our approach is designed for contemporary plug-in architectures such as those of today's web browsers *without needing any code changes* or *access to their source code.* It does not require an understanding of the semantics of (potentially complex) host/extension interface, nor does it make strong assumptions about the host application's ability to defend itself from attacks mounted by a plug-in (including attacks that simply involve calling host API functions with malicious arguments). Yet, it is strong enough to work against malicious plugins that employ active evasion techniques, and practical enough to secure many of today's plug-ins. Finally, it imposes only modest performance overheads in practice.

1.1 Challenges of Securing SMEs

The central challenge in extending application confinement mechanisms to SMEs is that of distinguishing the actions of the SME (which need to be confined) from those of the host application (which should not be confined, or else we risk breaking the host application). Such discrimination is hard because an SME can hide malicious activities by "tricking" its host application into doing their bidding. For instance, an SME may:

- subvert the control-flow of its host application by modifying the code pointers used by the host application, including function pointers, return addresses, etc.
- corrupt host application data, including file names, data buffers, etc. As a result, an SME can control which files are opened by the host, and what data is written to them.
- employ a slew of powerful stealth and evasion tactics that are available in a shared memory environment, e.g., incorporate memory errors that make it difficult to analyze or predict an SME's behavior, modify the data structures (say, the runtime stack) used to attribute security-sensitive operations to the plug-in or its host, incorporate and/or exploit concurrency bugs, etc.
- perform attacks that violate low-level assumptions made in binary code, e.g., modifying registers (or call-return semantics) in a manner that violates the platform-defined application-binary interface (ABI). For instance, a modification to the stack pointer will allow an SME to exert considerable control over the control flow within benign code after a return from SME.

Since the basis of many of the above attacks is shared memory, some research efforts such as XFI [11] have been built over memory isolation. Unfortunately, although memory isolation provides an important primitive for securing untrusted SMEs, it also negates many of the primary advantages of plug-in architectures. Specifically:

*This author is now at University of California, Berkeley

- SMEs are popular because they can exchange complex data structures easily. The host-extension interface can use aggregate data structures that contain multiple level of pointers to memory regions allocated at different points in the code of the extension or the host. Enforcing memory isolation require the scope of legitimate accesses (e.g., which pointers will be accessed, how many levels of indirection will be followed, etc.) to be identified in advance by a programmer. The host and/or SME code needs to be redesigned/modified so that all shareable data is allocated within shared memory regions, or is explicitly copied at the SME/host boundary. Moreover, any data residing in a shared region becomes susceptible to attacks by a malicious SME.

  The scope of this problem is magnified in the context of large and complex host-application APIs. For instance, browser plug-ins on Linux can access the Netscape plug-in API [4] as well as APIs provided by the C and C++ standard libraries, GUI libraries, etc.; altogether, a plug-in can access a few to several thousand functions.
- Memory isolation does not prevent attacks on host-API functions that involve maliciously-crafted arguments. The host API functions need to perform adequate sanity checks on their arguments to protect against these attacks. Given the large size of the host-API mentioned above, it becomes far too cumbersome to identify which functions can be safely exposed to untrusted extensions, and if so, with what parameter values.

  Moreover, existing host-extension APIs have been designed for flexibility and programmability, with the assumption that extensions are benign. Hence, for many API calls, it may be difficult to predict the security ramifications or identify ways to limit them.

It is clear from this discussion that in the context of complex and large host-API interfaces such as those used in web browsers, isolation-based approaches require significant effort to modify existing host applications and plug-ins in order to make them secure. Moreover, even after expending this effort, these modifications and validation checks can not easily be correlated with higher level security objectives, e.g., preventing SMEs from controlling data that is ultimately transmitted over the network.

For these reasons, our approach does not rely on isolation or validation checks at the SME/host boundary. Instead, we develop an *efficient* taint-tracking technique to label and track data "controlled" by an SME, and enforce confinement policies on any system-calls whose invocations or arguments are controlled in this way. Since system-call APIs are much smaller (consisting of a few hundred functions), and have been designed with security objectives in mind, specification and enforcement of policies at this interface is significantly simplified as compared to the host/SME interface. Moreover, our approach enables system-call (or information-flow) policies applicable to a stand-alone untrusted application to be reused for an SME providing the same functionality.

1.2 Contributions

- *Analysis of the range of threats posed by SMEs.* In Section 2, we present a comprehensive analysis of the different ways an SME can compromise the integrity of its host application. Previous dynamic information-flow tracking techniques, including those that applied taint-tracking to malware [39, 10], do not address this range of threats. In contrast, we develop taint-tracking techniques that are secure in this environment.
- *Secure attribution.* In Section 4, we present a secure technique for attributing the actions of an application to the SME or the host application. This attribution technique provides the basis for containment policies, described in Section 4.5. The salient features of our attribution technique are as follows.
  - We present an simple and elegant technique for attributing control flow context to the host or the SME.
  - We present a novel technique for defeating race condition attacks, where untrusted code attempts to exploit the time interval between data and metadata updates to corrupt taint-tracking. Our technique avoids the use of locks and to provide an efficient solution.
  - We provide an concise analysis of low-level evasion techniques that may be employed by malicious code. We describe several new techniques designed to defeat such evasion attacks.
- *Analysis of protection provided by secure attribution.* In Sections 4 and 5, we justify why our techniques can effectively mitigate the threats described in Section 2.
- *Efficient and robust binary instrumentation framework.* Unlike previous taint-tracking techniques that required source-code access [36] or relied on dynamic binary instrumentation techniques that have high overheads [22, 24], our approach is based on static binary rewriting. Moreover, we achieved this efficiency without making optimistic assumptions such as those made in [25] that do not hold in the context of untrusted SMEs.
- *Experimental evaluation of effectiveness.* We have built a prototype, called SafeBind, that embodies our approach. As described in Section 7, SafeBind is robust enough to deal with moderately large programs such as Firefox and Konqueror, and incurs modest overheads (10% to 30%) under realistic deployment scenarios.

  The downside of a conservative approach such as ours is the possibility of false positives. Our experiments show that for many commonly used plug-ins, our approach can avoid false positives.

2 Threat Model and Related Assumptions

In this paper, the term "untrusted SME" refers to plug-ins obtained from untrusted sources. We assume that only

the binary code for the host application and the SMEs are available. The SME may take the form of one or more libraries that are linked (either statically or dynamically) with the host application code to form an executable that runs within a user-level process.

Our goal is to defend the integrity[1] of a host system from an SME that purports to provide some benign functionality, but turns out to be malicious. We take a conservative approach that ensures that untrusted SMEs won't violate integrity, but in doing so, may have to reject some benign SMEs, e.g., an SME that incorporates code obfuscation that prevents disassembly[2]. Although our current implementation does not support dynamic code generation, it is relatively easy to do so: our instrumentation, which is currently performed statically, needs to be performed at runtime to newly generated code before it is run.

For the sake of concreteness, we assume that confinement policies will be stated in terms of system calls that can be accessed by the SME, the value and the taint associated with each of their parameters, and (in case of systems that support file integrity labels) the integrity labels of objects accessed by the system call. We assume that the goal of a malicious SME writer is to execute system calls that violate this policy. An SME may achieve this by "controlling" the flow of execution that invokes a violating system call, or by "controlling" the parameter values provided to a violating system call. (Being unable to identify operations effected by the host program on behalf of the SME leads to the commonly known "confused deputy" problem.) Our defense, based on taint-tracking, defeats such attacks by (a) using a conservative attribution technique that identifies any such "control" attempt; and (b) by preventing evasion and subversion of taint-tracking and attribution logic.

A malicious SME has the following choices in terms of possible approaches for evasion and/or subversion of higher-level integrity policies:

- *Subverting program control-flow*. Untrusted code may subvert program flow of the trusted host by corrupting function pointers, and return addresses to directly execute sensitive operations, or execute host code that in turn invokes sensitive operations. Moreover, corruption of its own code pointers may allow an SME 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, and `longjmp` to transfer control to unintended code.
- *Corrupting host application data.* An SME may directly corrupt host application data that is stored within the process address space. It may start from one of the pointers contained in the data structures passed by the host application to the SME, and traverse down several levels of pointers to identify candidate data structures that can be corrupted — such corruption attacks are easy in the context of APIs designed for benign extensions, (e.g., the Netscape API [4]) where the exchanged interface data structures contain several pointers to objects owned by the host application. Even if no useful data (or pointers) are explicitly passed into an SME, it may still be able to find critical host data structures based on knowledge about the location of global variables, or by scanning the stack or the heap. Finally, instead of using its own code to perform data corruption, an SME may utilize a host API function (or a snippet of host code) to do the actual copying.
- *Evading taint-tracking.* Dynamic taint analysis techniques can accurately reason about explicit flows, i.e., dataflows that take place via the assignment of a tainted value to a variable. However, data may flow as a result of *implicit flows* that cannot be detected without using static analysis. Unfortunately, it is difficult to perform implicit flow analysis in binaries due to their low-level nature, and the use of address arithmetic, pointer indirection, and so on. As a result, implicit flows have been ignored by previous dynamic taint analysis techniques, including those used in the context of malware analysis [10, 39]. Although this is acceptable in the context of trusted code, where it is reasonable to assume the absence of significant "covert channels" due to implicit flows, it is trivial for malicious code to use implicit flows for propagating large amounts of data. (See [33] or [6] for examples.) We therefore rely on a conservative approach that taints all writes within untrusted SME code, while making less conservative assumptions on the trusted host application code.
- *Circumvention or subversion of instrumentation logic.* An inline reference monitoring approach such as ours, where the instrumentation resides within the same address space as untrusted code, is subject to following types of attacks: (a) corruption of instrumentation data, (b) bypassing instrumentation code, (c) invalidating assumptions made by instrumentation code, and (d) exploiting program logic to confuse attribution.
  - *Metadata corruption.* With a taint-tracking based defense, a malicious SME can defeat detection if it modifies critical data structures while ensuring that the corresponding taint tags do not reflect this change.
    - *Direct metadata corruption.* To achieve this, an SME may attempt to overwrite the taint tags either

[1] Although information-flow based techniques such as the one developed in this paper can handle both confidentiality and integrity policies, our primary focus in this paper is on integrity. As a result, we don't handle SMEs that manipulate highly confidential data, such as an untrusted password manager extension to a browser.

[2] Malware analysis techniques do need to cope with obfuscation, since it is routinely employed by malware. However, our goals do no require support or analysis of malware, so our approach is designed to detect code features that can defeat our instrumentation and policy enforcement techniques, and mark the corresponding SME as unsafe.

directly, or by "tricking" the host code to overwrite the taint tags.

  - ∗ *Race-conditions.* With a binary instrumentation technique such as ours, there is a small window time between data updates and taint tag updates. This makes race condition attacks possible in a multi-threaded application. In a write/write race, an SME may race with a benign thread so that the data update reflects the data written by the SME thread, while the taint tags reflect the data written by the benign thread. In a read/write race, a benign thread reads the taint tags preceding a data update by an SME thread, while the data read reflects this update.

– *By-passing instrumentation code.* An SME may attempt to subvert instrumentation by jumping into the middle or the end of instrumentation code. Alternatively, it may attempt to return to a different location than that of the call, with the intent of bypassing additional checks that are inserted at the original return site to ensure safety of return values. This class of threats falls under the general class of *low-level control-flow integrity* threats.

– *Invalidating assumptions made by instrumentation code.* A binary instrumentation approach typically assumes that all of the code respects the application-binary interface (ABI), which specifies restrictions on register usage, function-calling conventions, etc. For example, it specifies that certain registers (called "callee saved registers") are left unmodified across a function call. Similarly, each thread is expected to have its own stack that is disjoint from other thread-stacks, and the global/heap memory. In an *ABI attack*, an SME violates these constraints in order to confuse the defense mechanisms, and/or to modify data (e.g., data in a callee-saved register) without being noticed by the instrumentation mechanism.

– *Exploiting program logic within trusted code to confuse action attribution.* Included in this class are attacks that exploit control dependences and implicit flows in trusted code.

3 Basic Instrumentation Techniques

3.1 Static Binary Instrumentation

The first step in instrumentation is that of disassembling a binary. Robust disassembly of so-called stripped binaries continues to be an active area of research. Kruegel et al [16] have described a combination of static analysis and statistical techniques that have been shown to be robust even in the presence of some degree of obfuscation in programs. Unfortunately, their techniques cannot guarantee accurate disassembly of all code. More recently, Nanda et al [21] developed a robust disassembly technique that is suitable for instrumenting binaries. Their approach uses a quasi-static disassembly approach, where most parts of an executable are disassembled (and instrumented) statically, while a small part of the code that cannot be statically disassembled is instrumented at runtime. This approach can work well with our technique, but since the focus of this paper is not on disassembly techniques, we have simplified our implementation task by assuming that the binary contains information about entry points of all functions, at which point the simpler techniques from [37, 26] can be employed. We do not assume the availability of any additional symbol table or debug information.

After disassembly, our technique constructs the control flow graph for each function, and records all its entry points. Next, it performs the actual instrumentation, introducing code for taint-tracking and other security checks. This instrumentation typically introduces one or more additional taint computation instructions for each instruction in the original binary. As a result, function bodies expand, requiring them to be relocated. If the code uses function pointers, they may continue to point to the original code version. Hence the original version cannot be deleted [21], but needs to be modified so that any target address that is reachable using an indirect control-flow transfer will now contain a jump to the corresponding location in the instrumented version. The rest of instructions in the original code are replaced with an invalid opcode in order to detect implementation bugs, as well as evasion attacks based on executing uninstrumented code.

Our instrumentation framework is designed to handle large COTS binaries such as those of Firefox and Konqueror browsers, and Apache server and modules. It is robust in the face of typical compiler optimizations such as frame pointer omission and tail calls, as well as hand-written assembly[3]. It can also handle position-independent code (PIC), C++ exceptions, UNIX signals, and so on, *without making many assumptions about the compilers involved*. Additional details on our static instrumentation techniques, including an explanation of how we handle these features, can be found in [25].

3.2 Instrumentation for Taint-Tracking

As done in some of the previous works on taint-tracking [36], we maintain the taint information in an array tag. For a location l, $tag[l]$ indicates if this location is tainted or not. Tag space could be allocated statically, or using an on-demand allocation as in [36]. We associate 8 bits of taint with each 32-bit word.

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 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

[3] Many popular applications such as Firefox and GIMP, as well as many media codecs, make use of hand-coded assembly.

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

add ebx, [ebp+0x1c]
```

Figure 1: Instrumentation for trusted code.

```
mov eax, VR1

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



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

mov VR1, eax

add ebx, [ebp+0x1c]
```

Figure 2: Instrumentation for untrusted code.

used for address computation (i.e., computing the location of $tag[l]$ from l) and taint-tag computation. Since virtual registers will ultimately be realized using memory, the instrumentation shown in Figure 1 uses them like a memory operand rather than a register operand. Virtual registers are saved in thread-specific storage that is accessed using standard OS conventions. (Unlike LIFT [24], we do not rely on the availability of unused processor registers for implementing taint-tracking; instead, our technique realizes virtual registers using main memory.)

Figure 1 shows the basic taint-tracking instrumentation for an instruction that adds the `ebx` register to memory location `ebx+0x1c`, leaving the result in memory. The first step is to save `eax`, `ecx` and CPU condition flags so that they could be used for computations in the instrumentation code. Then `eax` is used 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.

Since constants have a taint tag of zero, binary operations involving constants 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".

4 Secure Attribution and Policy Enforcement

In this section, we first develop a *secure control attribution* technique (see Section 4.1) to classify the current control flow to one of three contexts: (a) plug-in, (b) host application, or (c) a host-application function called by a plug-in. We then develop a *secure data attribution* technique that states whether the value of a data item (e.g., a system call parameter) is entirely under the control of the host, or has been significantly influenced by the SME. We begin by summarizing the differences between taint-tracking for benign and untrusted code. Following this, we address evasion attacks outlined in Section 2, including metadata races (Section 4.3) and low-level evasion attacks (Section 4.4). As described in Section 4.5, different *system-call based sandboxing* policies can be enforced based on this attribution, thereby enabling SME operations to be sandboxed without having to restrict operations being performed by the host application.

4.1 Instrumentation for Control Attribution

One obvious technique for control attribution is to examine the return addresses on the stack. Unfortunately, this technique is insecure since a malicious SME can corrupt or spoof stack contents. Although secure attribution techniques have been developed in the context of Java, this relies on the type-safety of the language. In contrast, we develop a technique for secure attribution on COTS binaries. Our technique uses two context flags C_t and C_u as follows:

- C_t is set whenever the instruction currently being executed is within the body of a trusted function.
- C_u is set whenever the current control flow is directly determined by untrusted code, e.g., when untrusted code is currently active on the runtime stack.

We use a simple and elegant instrumentation technique to update C_t and C_u that avoids runtime operations to scan the stack or to determine whether callee addresses belong to trusted or untrusted code. This simplifies the instrumentation and makes it efficient. Specifically:

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

Note that untrusted code can directly exercise control over execution flow by executing its own instructions or by calling other functions within trusted or untrusted code. In these cases, it is clear that C_u will be set. It can indirectly control the flow of execution by corrupting a function pointer used by the trusted code. From the above description, it is clear that C_u will be set in this case as well. Other ways for untrusted code to directly control the flow of execution are: (a) corruption of return addresses

used by trusted code, (b) jumping past instrumentation, (c) jumping into runtime generated code (which has not been instrumented), and (d) using exceptions and/or signals to effect control-flow transfers. All of these attempts are prevented by control-flow integrity checks described in Section 4.4. As a result, we conclude that the above instrumentation can be used to securely attribute current control-flow context of any thread:

- $C_t = false, C_u = true$: execution is entirely under the control of untrusted code
- $C_t = true, C_u = false$: execution is entirely under the control of trusted code, and
- $C_t = true, C_u = true$: trusted code is executing on behalf of untrusted code.

C_u and C_t are accessible within policies, thus enabling different policies to be enforced in different contexts. They are stored in thread specific memory, and is protected from direct corruption, much like the other data structures used by the instrumented code.

4.2 Taint-Tracking in Untrusted Code

The primary basis for control and data attribution is taint-tracking. The instrumentation for untrusted code performs "taint propagation" as described in Section 3.2. The source of taint is all data written by the untrusted code. Note that this conservative approach is more appropriate for untrusted code as compared to other choices, e.g., treating constants as untainted. An approach that treats constants as untainted would not detect an attack where security-critical data is overwritten by a malicious SME with a constant value, e.g., a variable with a value `/bin/login` is overwritten by `/bin/sh`.

As shown in Figure 2, the instrumentation for untrusted code is simpler than that of trusted code: any write by the untrusted code causes the corresponding taint tag to be set. Thus, the constant "1" is moved into the corresponding tag location instead of performing any taint computations. No metadata accesses are needed for read operations performed by untrusted code.

We note that the conservative approach of tainting all writes by SME effectively thwarts any attempts to evade dynamic taint analysis using implicit flows. In particular, it does not matter whether data written by untrusted code is implicitly or explicitly dependent on tainted data (or even independent of tainted data), as data written by untrusted SMEs is always marked tainted.

4.3 Instrumentation to Handle Metadata Races

The instrumentation described above performs data and metadata updates in separate instructions. If two concurrent threads B ("benign") and M ("malicious") update the same data, it is possible that M's data update will occur after B's, while M's metadata update precedes that of B. As a result of such a write/write race, the data and its associated metadata could be out of sync. Similarly, there can be a read/write race, where B reads metadata just before it is updated by M, but reads the data updated by M.

The narrow window of time between data and metadata updates makes the likelihood of successful race attack to be very low. However, malicious threads may introduce data races on purpose in order to exploit metadata races. By repeatedly racing with a benign thread, a malicious thread may be able to increase the probability of a successful attack to a considerable value.

An obvious approach to eliminate such races is to use locks to ensure that data and metadata updates occur atomically. However, given that every memory update involves a metadata update, such an approach will have a major performance impact. We have therefore developed a new *lock-free technique* for to address data/metadata races. Our technique is based on the following assumptions:

- If a race condition leads to benign data being labeled as tainted, that is acceptable. However, tainted data should never be labeled benign. The latter condition is identified as the "dangerous condition" DC in the discussion below. The reasoning for allowing the former condition is that in the context of non-malicious SMEs, data/metadata races should be rare; and if they do occur, then they lead to a denial-of-service rather than an integrity violation.
- Races involving multiple benign threads are not exploitable by an SME. The reasoning here is that a data-metadata race implies a race condition on the data involved in the access. Such race conditions lead to erroneous (or unpredictable) behavior, and hence benign code will typically incorporate some logic to avoid them. While it is possible that some race conditions may still be present in production code, they are likely to be rare, and moreover, it is unclear that an SME can intentionally exercise them. Hence we don't consider multi-way races involving multiple benign threads.

The first key idea in our lock-free technique is to perform data and metadata updates in different orders for read and write operations within benign code: (1) read operations will read the metadata after the data read, while (2) write operations will write metadata before the data. The second key idea is that (3) within untrusted code, metadata will be updated once before a data write, and then again after the data write. We now argue that these techniques eliminate the dangerous condition DC identified above.

- **Write-Write races**. In this case, both the benign thread B, and the untrusted thread U, write to the same memory location. If both threads write "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. For DC to occur, (i) U's data write must follow that of B. Note that

our instrumentation (2) ensures that (ii) B's metadata update will precede its data update, and instrumentation (3) ensures that (iii) U will update metadata once after its data update. Putting together the ordering constraints (i) through (iii), it is easy to see that whenever U's data update follows that of B, at least one of U's metadata updates will follow that of B, thus avoiding DC.

- **Read-Write races**. Since untrusted code never *reads* metadata, such a race condition involves a write operation by U and a read operation by B. By (3), U updates metadata with a "1" before and after its data write. So, the only possible way for DC to occur is if (i) B's metadata read precedes the first metadata update by U, and (ii) B's data read follows U's data update. However, this is not possible as our instrumentation (1) ensures that B's metadata read will follow its data read, and hence either (i) or (ii) cannot hold, thus avoiding DC.[4]

Thus, we conclude that DC can never arise, ensuring that our taint-tracking instrumentation is safe against race attacks perpetrated by an SME thread.

4.4 Defending Against Low-Level Evasion Attacks

4.4.1 Control-Flow Integrity (CFI) Restrictions

A malicious SME may attempt to evade our policies by jumping past the instrumentation code that enforces these policies, or updates data (such as the C_u and C_t flags or the taint information) used in these policies. To prevent this, we define and enforce the following CFI criteria.

CFI criteria for untrusted code. Our instrumentation enforces the following CFI properties:

- All intra-procedure transfers of control are to the beginning of one of the basic blocks in the same procedure. Attacks involving jumps to the middle of instructions, or to instructions inserted during instrumentation are thus thwarted. In addition, during the instrumentation, the absence of implicit control-flow transfer instruction (such as software interrupts) is verified.
- All inter-procedural control-flow transfers are to valid function entry points, which are required to be aligned as specified by the ABI.
- All return instructions return to legitimate return points, although the calls and returns need not match. Returns that cross trust boundaries require additional care, and are described separately.

[4]We did not consider the case where the taint was "1" after the pre-write metadata update by U, but became "0" before B read it. For this to happen, a third thread must have updated the metadata, and moreover, this thread should be benign as a "0" can be written only by benign threads. Thus, for this condition to occur, there should be a three-way race involving two benign threads and a malicious thread. However, this conflicts with our assumption that races involving multiple benign threads are either not present or not exploitable.

CFI criteria for trusted code.

- Direct control-flow transfers are not checked within trusted code: they are assumed to be satisfied since we trust this code, and expect that it was compiled with a benign compiler.
- For indirect control transfers, the instrumentation ensures that the pointer involved in indirection is untainted, or otherwise C_u is marked set. Moreover, it is ensured that the transfer goes to the beginning of some function within trusted or untrusted code.
- Return addresses should not be tainted — this can happen only due to corruption by untrusted code, and hence execution is aborted if this is detected.

CFI criteria for SME/Host interface. All control transfers across the host-extension interface require a stricter enforcement. We detect all such transfers either during our static transformation, or using runtime range checks on the control pointers used in indirect control transfer instructions and returns.

For cross-interface control transfer, SafeBind ensures that calls and returns match, and that the ESP is left restored across the interface. This is to prevent the untrusted code from using the return address or the ESP to arbitrarily choose its control transfer location. Note that the taint associated with the return address and ESP can not be trusted. For instance, when control returns from a trusted routine back to an untrusted routine, since the return address will always be tainted. The same is true for ESP when control return from untrusted code to trusted code.

To ensure that such critical state is preserved across interface procedure calls, our enforcement uses an auxiliary protected stack which strictly copies these values to and from the main stack. This auxiliary stack also forms the basis of other ABI conformance described subsequently.

CFI criteria regarding exceptional flows. SafeBind also deals with signals, setjmp/longjmp in C, and C++ exceptions (which uses setjmp/longjmp based implementation on our platform). SafeBind trusts all signal handler registration made in the trusted code, but ensures that untrusted code registers valid function start addresses in its code as signal handlers. This is sufficient to ensure that the attribution for control using C_u and C_t works as expected. Setjmp/Longjmp are C functions and do not require special handling. The attack that involves corrupting the control pointer used by longjmp is handled by our CFI restrictions — untrusted code can only return to valid return points when using longjmp. As indicated in Section 4.1, this will immediate set the C_u flag. Similarly, such an attack will disallow arbitrary control transfers into points in trusted code.

Instrumentation for enforcing CFI criteria. When the control-flow target is statically known, enforcement amounts to a check that is performed at instrumentation

time. Otherwise, instructions are inserted into the code to perform this check. To implement this, we make use of a bit-valued array CFT that is indexed by code address. $CFT[A]$ is set iff A is a valid control flow target. Since code addresses are aligned on 4-bytes on most systems, CFT array will only require $1/32$th of the total code size.

4.4.2 Runtime ABI-conformance checks

Similar to CFI, ABI semantics is assumed to be preserved by trusted code, but is explicitly enforced on untrusted code. As per our tainting technique, it is simple to see that violation of most ABI conventions, such as ensuring that "callee-save" registers are left unchanged, are harmless when they happen entirely within untrusted code. However, ABI semantics needs to be *explicitly* enforced when control transfer takes place from a trusted to untrusted context and then back. This is ensured using additional instrumentation at points where trusted code calls untrusted code. This instrumentation explicitly saves callee-save registers before the call and restores them afterwards. Note that the alternative of relying on the taint status of callee-saved registers does not work: since they are saved and restored by untrusted code, their values would always be tainted, even when the original values assigned to them by trusted code are preserved. As described above, a protected auxiliary stack is used for saving these registers.

As part of ABI requirement, runtime validity checks are performed on ESP before calls from untrusted to trusted code — specifically, on the x86 we check that ESP is in the thread's stack region and is below the value at the time of last entry in the untrusted code. It should also be clear that the protected auxiliary stack ensures that ESP is left restored for all returns from untrusted code explicitly, disallowing usage of such callee-saved registers as means to violate integrity in trusted callers.

We point out that previous works [10, 39] don't treat the full range of attacks against attribution mechanisms, and are hence vulnerable to some of the above attack avenues.

4.5 Specifying and Enforcing Security Policies

In our framework, security policies can be enforced at the point of invocation of any function within the host system. A security policy is a predicate with the following inputs:

- the function being invoked and its parameters[5]
- the taint associated with each of the parameters
- the control attribution flags C_u and C_t.

While it is possible to develop a high-level policy language that takes these inputs, that is not the goal of this paper. We assume that high-level policies may be compiled into a piece of code, and provide the ability to interpose this code before and/or after the call to each host function for which a policy is specified.

[5] If the function uses global variables that are relevant for policy enforcement, then, from the perspective of policies, they too are considered as parameters to the function.

We believe that the framework presented so far can support a range of security policies for confining untrusted SMEs, while providing a level of flexibility, power, and ease of policy development that is similar to previous works on securing stand-alone untrusted applications. However, for brevity, we only discuss system-call policies below. A system call can be attributed to an extension if C_u is set, or if any of the arguments to the system call are tainted. System calls attributed to the host are unconstrained, whereas a specified system-call policy can be enforced on the rest. Moreover, if a sandboxing policy is available for a stand-alone version of the extension that offers the same function, we can reuse the same policy.

This basic approach can be further refined, e.g., to ignore taint on system-call arguments that don't impact security, or to attribute certain system calls to the host when both C_u and C_t are set. Moreover, if some extension API functions are known to perform adequate input validation, then we can untaint the arguments to such functions at the point of call. (This is often referred to as *endorsement*.)

5 Effectiveness Against Threat Model

We now analyze the techniques described in the previous section with respect to the threats described in Section 2.

- *Subverting program control-flow*. This was already addressed in Section 4.1.
- *Corrupting host application data.* 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. Moreover, if an SME uses a host function to copy some data into its intended target data structure, it needs to pass in the location of this destination using a parameter, or by modifying a global variable used by the host. In either case, our taint-tracking technique will mark the destination as tainted, thus ensuring that in all cases, a security policy based on taintedness of system call arguments cannot be subverted by corrupting host application data.
- *Evading taint-tracking.* We already pointed out that evasion techniques such as the use of implicit flows cannot thwart our conservative tainting technique that marks all data written by untrusted code as tainted.
- *Circumvention or subversion of instrumentation logic.*
  - *Metadata corruption.*
    - *Direct metadata corruption.* This is prevented using the technique described in [36][6]. This technique can be used to protect all metadata, including the tagmap and all other data used by the instru-

[6] Note that any instructions in (trusted or untrusted) code that writes into any memory m will be preceded by an instruction that updates $tag[m]$. If $tag[l]$ is left unmapped for all locations l that we want to protect from direct access by the (trusted or untrusted) code, then any such access will cause a memory exception, causing the program to be aborted.

mentation code, such as the C_u and C_t tags.

∗ *Race-conditions.* We previously established in Section 4.3 that metadata races cannot be exploited to defeat our technique.

– *By-passing instrumentation code.* The control-flow integrity checks described in Section 4.4 were specifically designed to defeat these attacks.
– *Invalidating assumptions made by instrumentation code.* Our ABI enforcement techniques (Section 4.4) were designed to address these threats.
– *Exploiting program logic within trusted code to confuse action attribution.* The most powerful subversion mechanism involves corrupting pointers (or array indices) used by trusted code so that they point to data sources or destinations chosen by the SME. This mechanism is already thwarted by our current technique, since it marks any data read or written using a tainted pointer to be tainted. This leaves only conditional dependences and implicit flows within trusted code as the only means for evading data attribution. While one cannot rule out the possibility that these may be exploited, we point out that existing work on taint-based attack detection [22, 36, 24] does not consider this covert channel to pose a significant threat *since we are dealing with trusted code.* Moreover, this avenue requires attackers to find *exploitable* control dependences or implicit flows within trusted code, and craft an attack to exploit them. To further limit the attacker's choices we plan to consider incorporating limited forms of control dependence tracking on benign code. (However, we do not consider implicit flows within trusted code as a serious threat.)

6 Optimization

Performance is critical for realizing a practical system that relies on heavy instrumentation such as fine-grained taint tracking. Instrumentation for taint propagation is the most significant factor, and hence is the focus of our optimization techniques. The best known overheads for binary-based taint-tracking on CPU-intensive benchmarks has been achieved in our previous work [25] (90% to 180%). However, this performance is obtained using optimizations that are not sound in the context of untrusted code. In particular, it relies on a number of optimizations that assume that local variables of one procedure won't be accessed by another procedure unless their addresses are explicitly passed as parameters. A malicious SME can defeat the taint-tracking mechanism by intentionally violating this assumption. We have therefore developed alternative optimizations, as described below, that were explicitly designed to be sound in the face of untrusted code.

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

Figure 3: Instrumented code after liveness optimization.

```
movd eax, xmm0
add ebx, edx
cmp ebx, 0
movb 0,
  [ebp+0x801c]
mov ebx,
  [ebp+0x1c]
movd xmm0, eax
je 0x40000
```

Figure 4: *Fastpath* version.

6.1 Low-level Optimizations

Inline code-instrumentation requires maintaining different execution contexts between application and taint-tracking code. Additional registers to perform instrumentation-related computations need to be saved prior to each instrumentation snippet and restored afterwards resulting in expensive *context switches*. The basic taint instrumentation adds 10 to 20 instructions for each instruction in the original code that needs taint-tracking. Worse, about 10 additional memory references are added for each original memory reference. To improve performance, we developed the following optimizations.

- *Reducing register usage through instruction selection.*
  – *Using CPU flags to perform taint computation.* Our initial instrumentation in Figure 1 needed 3 physical registers for realizing the virtual registers, plus another register to hold a pointer to the thread-specific register taint data. We reduced this by 1 register by using the CPU flag register for intermediate taint computation.
  – *Packing register taint and saved CPU flags into one register.* By packing the taint for all CPU registers into a 8-bit quantity, and using the remaining bits in a 32-bit register for saving CPU flags, we further reduced the number of registers to just 2.
- *Reducing memory accesses by using rarely used XMM registers.* XMM (eXtended Multi-Media) registers are unused in most programs, so we utilized them as a scratchpad for saving general-purpose registers needed for computation. Although XMM registers do not provide a significant performance boost over L1-cache (which approximates the speed of accessing frequently used memory locations), they are a win in multi-threading code because these registers are thread-specific, and thus eliminate the virtual register needed to hold the base of thread-specific store. When XMM

registers are used by the application itself, we resort to memory resident scratchpads. thread-specific store.

6.2 Higher Level Optimizations

In this section, we describe several higher level optimizations that we have adapted from previous works to ensure that they are sound in the context of untrusted code. We have implemented them, and have obtained significant performance boost as a result of these optimizations.

Liveness optimizations This optimization is aimed at reducing the context switch overhead by improving the selection of physical registers that are used as virtual registers. It is conceptually similar to those used in previous works such as [24]. However, as compared to dynamic rewriting systems, our liveness analysis is more efficient as it is applied across basic blocks.

Currently, we have used a simple strategy — 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 need to be saved only once per sub-block instead of once every instruction. We also eliminate need for saving and restoring conditional code register when it is unused subsequently.

After the low-level optimizations and liveness optimization, the instrumented code size comes down as shown in Figure 3. In this figure, the instructions are shown in bold-face underlined are the original instructions, and the rest correspond to our instrumentation. The instrumentation is simplified for ease of understanding, eliminating out-of-order tainting for preventing races, and ignoring extra instructions for misaligned accesses. Also note that we cover only the taint-tracking instrumentation here, and don't show CFI, ABI or other instrumentations.

Generating multiple code versions Similar to [24], and as described in [25], we generate a *fastpath* and *slowpath* versions of the code for each function. The slowpath version is unchanged from before the optimization. The fastpath assumes that all registers are untainted, which means that the output of every instruction will be untainted as well. As a result, taint-related instrumentation can be avoided on the fastpath except for memory loads (where a check for taintedness is made and control transferred to the slowpath version in that case) and memory stores (where a zero value is stored into tag.) Figure 4 illustrates the fastpath optimization.

7 Implementation and Evaluation

SafeBind currently works on Ubuntu Linux desktop environments. Much like previous static transformation systems [17, 37], it has two major subcomponents — a binary analysis subcomponent and a static transformation subcomponent. The analysis subcomponent is written from scratch using C++, uses Intel's XED library component from Pin [19] in its x86 specific backend for decoding. The transformation subcomponent uses LEEL [37] for ELF editing, XED for higher level information about operand usage, and `nasm` assembler for encoding.

We evaluated our system in a practical online deployment setting on typical host applications such as web browsers and web servers. All our experiments were conducted on an Intel Pentium M equipped with 1.6GHz processor, 512MB RAM, and running Linux kernel version 2.6.17. We tested our transformation system systematically with a series of tests on variety of programs ranging from Linux utilities like `cp`, `gzip` to large applications such as `gimp-2.2`, `gaim`, `pdftops`, `vlc` `xmms`, Firefox (56K functions, 5 MLOC) and all the needed libraries of the KDE 3.5.6 platform (over 2 MLOC).

7.1 Effectiveness

The primary goal of these experiments was to show that it is relatively easy to apply well-known policies for standalone 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 the host performs certain actions on behalf of the plugin. We believe that our approach is applicable to popular web applications such as browsers, email clients allow multiple extension mechanisms, given nearly all these applications allow full binary code execution via shared library plugins. To give an estimate of the extent of usage of SMEs, we point out that for Mozilla Firefox on Windows platform alone, there are 130 MIME types supported, with 78 plugins to handle these mime types. We were able to identify (by mere inspection of the shipped package or public package desciptions) that at least 52 new shared libraries could be attributed to these packages, not considering Firefox "extensions" that may contain binary code in addition to scripts written in other sandboxed languages such as Javascript. Our primary target in policy enforcement experiments were two popular web browsers – Konqueror and Mozilla Firefox, and a web server. We show experiments on plugins of different sizes and functionality.

Konqueror `kpdf` PDF viewer plugin. We considered the plugin version of the core of the KPDF viewer *libkpdfpart.so* as untrusted. We used existing policies for a standalone document viewer that were developed in the context of our model-carrying code work [28, 27]. This policy allows arbitrary file reads, while restricting file writes to a small set of files that were "owned" by the application (e.g., KPDF preference files). This policy does not permit the extension to make any network reads and writes, which achieves the intended goals of operation in the browser. 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

Frame Rate	Standard	CPU Overhead (%)
24 fps	Cinema film	10.7
25 fps	PAL progressive	10.8
29.97 fps	NTSC progressive	10.8
50 fps	PAL	19.9
59.94 fps	NTSC	28.1
60 fps	Monitor framerate	30.1

Figure 6: Summary of Micro Test benchmarks for VLC media player transformed as untrusted

a result, browser functionality wasn't restricted in any way by this policy.

Firefox with VLC media player plug-in. We used a policy developed in [23] for a stand-alone media player ("km-player") to the VLC plug-in. 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, with the exception of its preference files.

Firefox with only MPEG decoding library. Libmpeg is used for decoding mpeg images and is widely used on desktop applications. We used this in conjunction with Firefox to view streaming video online. We allowed the library to make connection to the display server, and restricted all data from the library to a single external interface for display, namely, a unix domain socket connecting to the X server. No other tainted data was allowed in any other system calls. The normal functioning of the browser (as well as that of MPEG library) was unaffected.

Firefox with ALSA sound library. We used a commonly used sound library libalsa.so on Linux, which is shared by many web applications running in the browser for streaming audio. We restricted it to use the sound device interface on our system, and allowed read/writes to this interface. Once again, the policy could be enforced without impairing the functionality of any component.

Apache `mod_config_log` module. In this experiment we took the case of another extension that is less trusted than its host system, specifically, a logging extension of the Apache web server that logs web server requests files in a specified ("logs") directory. As a policy for this plugin, we restrict writes of data controlled by this module to files in the "logs" directory only.

7.2 Performance

Our performance tests were focussed on evaluating overheads observed under realistic deployment scenarios for plug-ins. Instrumentation times were roughly comparable to the build times of the components transformed.

Apache. The trasnformed Apache server was connected to a client machine by a 100 Mbps link. We configured 100 clients to query web pages of different sizes using WebStone 2.5 benchmark. When the size of web pages was increased from 2K to 16K, the overheads decreased from a maximum of 28% to 0.6% on slowpath, and from 7% to 0.2% on the fast path, averaged over 5 runs. This is because as page size increases the server becomes more IO bound than CPU bound. The code size overhead is higher for the fastpath version because two versions of the code are generated.

Firefox. For this test, we used a benchmark tool from Mozilla Corp. used internally for performance testing. It uses a script that displays 350 web pages sequentially. The web pages are selected to including various features of the web page contents such as CSS, JavaScript, images, animations and so on. We measured the native CPU overheads, and as in the case of Apache, overheads for the optimized slowpath and fastpath versions of transformed Firefox code. On optimized slowpath, we measured a CPU overhead of 17.3 % while on the fastpath version of the code we measured 6.1 %, averaged over 4 runs.

Movie Player. We ran VLC media player, configured to not drop any late frames, in order to measure if there was any latency in viewing a movie file. We instrumented VLC as untrusted, along with with all the mpeg codec libraries. The elapsed time of the test movie files was unaffected, and no perceivable deterioration video quality was observed. To confirm this, we performed a series of micro benchmarks to measure the interframe latency for the VLC player for commonly used frame rates summarized in 7.2. In all cases, the elapsed time of the samples remained unchanged. Both the mean interframe latency (speed of video delivery) and the frame jitter (measured by the standard deviation of interframe latencies) showed almost no change after instrumentation – at most 1%(and often negative) increase, which could be due to experimental error.

For untrusted transformation on VLC player, we measured a CPU overhead increasing from 10.7% (24 fps file) to 30.1% (60 fps file) as the frame rate increases, with an average of 18.4 % slowdown over the uninstrumented version over 6 runs. The code size increased by a factor of 2.74x.

7.3 Defense against malware

Experimental evaluation of the defensive capabilities of SafeBind against real-world malware is complicated by two factors. First, due to our choice of platform, namely, Linux, we could not find any shared-memory malware. We addressed this complication by using stand-alone malware that is available on Linux, and packaging them into plug-ins. The second (and more important) complication is that existing malware has been developed in an environment where there are no practical defenses against shared-memory threats. Hence such malware deploy techniques that can be very easily detected, and hence do not satis-

Apache Results						
	2K	3K	4K	8K	16K	$\Delta CodeSize$
Slowpath	28.5%	13.9%	0.8%	1.1%	0.6%	6.8x
FastPath	7.1%	3.6%	0.0%	0.5%	0.2%	10.3x

Firefox Results		
	CPU Ov.	$\Delta CodeSize$
Slowpath	17.3%	7.1x
FastPath	6.1 %	10.5x

Figure 5: CPU Overheads and code size increase for (Left) Apache : Server Throughput Reduction on Apache web server for various web page sizes (in K) and total code size increase. (Right) Firefox : CPU Performance for loading 350 pages from disc, and total code size increase.

factorily test the capabilities of our defense. We address this complication by developing malware with features designed to evade our defenses.

We point out that the above complications in malware evaluation should not be a reason for concern. The confidence in defense against malware should be based on the principled analyses provided in the paper, rather than tests involving malware samples that simply do not incorporate any evasion techniques. At best, these tests involving malware simply provide another test case that provides evidence regarding the correctness (and robustness) of our implementation. For this reason, the emphasis of our evaluation has been on usability and performance of benign plug-ins rather than malicious ones.

Specifically, for evaluation, we used existing malware of other kinds and ported them as a PDF viewer Firefox plugin. We set a default policy for several plugins disallowing network activity, allowing file I/O to PDF viewer's owned files, and disallowing any tainted system calls in trusted context outside the untrusted scope. We tested 8 malware from [1] which consists over existing Linux backdoors and rootkits, and SafeBind was able to detect all these malware. For instance, malware such as Blackhole attempts to execute `execve` a shell prompt which is prohibited by our default policy. Similarly, user level rootkit functionality displayed by malware such as bobkit, tuxkit and lrk5 was detected by SafeBind, disallowing the actions to overwrite system utility files such as `/bin/ps` outside the policy's allowed domain.

In addition to these, we evaluated our defense against other techniques that are common in malware, or related to plugins we consider by crafting or own integrity violating "malware". Specifically, the techniques we employed:

- Subversion by overwriting the .dtor section entries, function pointer overwrites, embedding exploits in code such as buffer overflows/heap overflows, making network calls to attacker's remote host to fetch data in host buffers, overwriting critical system files and preference files. SafeBind was able to thwart all of these stealthy attacks that take advantage of the large shared address space.
- Self-modifying code and Unpacking code

  Note, by default our mechanism prevents execution of code generated on the fly or any self-modifications to code, since only code that is disassembled is allowed to be executed. This policy thwarted several existing malware samples that employ disassembly techniques, unpacking techniques and so on. SafeBind correctly detected these and flagged them as malicious
- Plugins injecting code from remote users.

  Several plugins such as Flash player and PDF viewer have recently been subjected to untrusted inputs that cause them to be corrupted and behave maliciously. In many such attacks, code injection is a first step which is thwarted by our default policy. Certain malicious plugins may purport benign intent, hiding such malicious behavior [6] until a remote attacker uses them for its own controlled action. SafeBind successfully thwarts such malware.

7.4 False Positives

In many of the experiments described in the above section, SafeBind incurs no false positives, i.e., none of the system calls made in the trusted context had tainted arguments. For instance, with Firefox/VLC we observed no false positives. This is because these plug-ins primarily read input data (or parameters) from their host, but their subsequent actions simply involve making system calls themselves, without involving Firefox code. Some false positives were initially observed with Firefox/PDF, Firefox/ALSA and Firefox/MPEG. On closer examination, we found that these involved certain benign system calls such as gettimeofday, sched_setaffinity, and read/write operations involving files opened by the plug-in. Hence we relaxed the policy to permit these operations.

To a certain extent, the lack of false positives is explained by the fact that dataflows in the higher level host code are not closely intertwined with the dataflows in the plug-in. Instead, most dataflows involve the plug-in and the general purpose (but still trusted) libraries provided by the platform, such as the C++ library, glibc, KDE library and so on. As a result, we observe tainted data being processed in library calls made by untrusted code, but otherwise we don't often see the use of tainted data. For plug-ins that involve close interactions with the browser code, there is reason to expect that there will be more false positives. They will need to be addressed using a combination of validation checks and endorsements (see below), and

refinement of policies (e.g., gettimeofday example.)

There was a point when we did experience some false positives due to `malloc`, when a buffer freed by untrusted code is subsequently returned by an malloc-call made by trusted code. To eliminate this false positive, it is necessary to develop a wrapper for `free` (and related routines) that first checks its argument for validity (i.e., it was a buffer that was previously allocated and has not yet been freed), and the untaints the pointer before returning it to the free list. The discussion above reflects the false positives observed after the inclusion of such a wrapper. It is easy to see that reuse of shared resources (or buffers) across trusted and untrusted code can lead to false positives in general, and need to be handled using such endorsements. Our experience to data provides evidence that need for manual effort for developing such wrappers will be relatively infrequent.

8 Related Work

Securing Extensions SFI[35] is a language-based technique to provide higher performance memory protection at finer granularity than is possible with OS-based techniques. Nooks[32] and follow-on works such SafeDrive [40] aim do memory region based policies at different granularity, but discovering these regions can be quite involved when dealing with complex data structures. Their work target is mainly kernel extensions, and the focus is fault isolation and recovery from error-prone extensions, but not protection from malicious ones.

The closest research to our work is XFI [11], which builds on some of the ideas from SFI and control-flow integrity (CFI) [2]. It is designed to work in user space as well as kernel space. It targets applications where there exists a narrow interface between trusted and untrusted code, with memory sharing limited to a few contiguous ranges. It handles some of the low-level attacks that our technique aims to handle. However, SafeBind shows how to achieve high-level security objectives on simple as well as complex extensions (e.g., browser plug-ins) that can make system calls, or utilize large system libraries were not considered by them. Second, our goals are different from isolation based techniques – we address the practical problems that arise when there needs to be sharing. Defining XFI policies for memory sharing, and for safe usage of the large APIs used by such complex extensions can be very cumbersome as opposed to policies at the narrow system call interface. In contrast, we have shown that the approach developed in this paper can be applied to complex extensions with relative ease. Moreover, our taint-tracking approach makes more realistic assumptions about vulnerabilities in trusted code — rather than assuming that it can defend against arbitrary shared memory changes (or input parameters provided to trusted functions), our approach allows data provided by untrusted code to be treated with adequate caution by tracking its flow during execution. These arguments also hold true against other extension isolation mechanisms such as Palladium [8].Recent works [9, 13] have proposed redesigning the browser using isolation techniques ranging from virtualization to process separation. Our technique based on automatic rewriting and require no modifications to the host, making them applicable to existing popular web browsers and give strong guarantees even in cases of large sharing.

Taint Tracking Information flow has been a topic of research for a long time. Most research in the past decade has been on static analysis based information-flow techniques [20, 29, 14] that achieves non-interference [12]. Practical application of these techniques have required significant programmer effort in the form program annotations, and hence aren't very practical for large-scale systems.

Fine-grained dynamic taint analysis has emerged recently as a practical alternative to static-analysis based information flow techniques. By focusing on explicit flows that take place through assignments and by largely ignoring control dependences and implicit flows, these techniques have avoided the need for manual annotations. However, they have suffered from significant performance problems [22], or required architectural support [7, 30]. The performance was improved significantly using a source-code transformation [36] instead of a binary transformation.

There has been several recent works on using taint-tracking to detect malware [10, 39]. In contrast to previous work, our techniques addresses the threat model and is designed to defend against an adaptive malware author. More recently, there has been offline analysis of hooking behavior of malware[38] that does account for some attacks perpetrated by the malware. However, these techniques have focussed on offline behavior monitoring and face different challenges when dealing with the adversary than ours. SafeBind is designed to be *efficient* in an online setting, and its goal is to allow benign extensions to operate while giving strong guarantees against host integrity corruption.

More recently, [24] et al improved the performance of binary taint tracking significantly. They rely on dynamic binary translation rather than the static transformation approach used in our approach. This has enabled us to develop several static analysis based optimizations that have yielded significantly better performance than that reported by them, improving the performance by a factor of two or more. Another important improvement in our approach is that it is multi-thread safe. Our previous work [25] has provided considerable improvements in performance, but it relies on optimizations that are unsafe in the context of untrusted SMEs.

9 Conclusion

In this paper we presented a new technique for securing untrusted software extensions. Our technique is able to support both simple and complex extensions such as browser plug-ins that use complex data structures and a very large API to interact with its host application. We developed a new static binary transformation technique for fine-grained taint-tracking, and presented several effective optimization techniques to improve its performance. In spite of the powerful adversary model presented by a malicious shared memory extension, we presented a systematic analysis to support our claim that they can be selectively confined using security policies that are similar to those used for stand-alone applications providing the same functionality. Our techniques are robust enough to be used on large applications such as the Firefox and Konqueror browsers. They enable effective enforcement of simple security policies to provide concrete assurances about system integrity from untrusted plug-ins, without unduly restricting the functionality of host application code. Our experimental results suggest that performance overheads can be kept low (about 20%) in realistic deployment scenarios for these plug-ins, thus establishing the practical value of the techniques presented in this paper.

References

[1] http://packetstormsecurity.org/UNIX/penetration/rootkits/index2.html.

[2] 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.

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

[4] Netscape Plugin API. http://developer.mozilla.org/en/docs/Plugins.

[5] K. J. Biba. Integrity considerations for secure computer systems. In *Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts*, 1977.

[6] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. Technical report, Secure Systems Lab at Stony Brook University, 2007.

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

[8] Tzi cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, 1999.

[9] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy*, 2006.

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

[11] 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.

[12] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[13] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with op web browser. In *IEEE Symposium on Security and Privacy*, 2008.

[14] Katia Hristova, Tom Rothamel, Yanhong A. Liu, and Scott D. Stoller. Efficient type inference for secure information flow. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, SIGPLAN Notices. ACM Press, June 2006.

[15] Francis Hsu, Thomas Ristenpart, and Hao Chen. Back to the future: A framework for automatic malware removal and system repair. In *Annual Computer Security Applications Conference (ACSAC)*, December 2006.

[16] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. 2004.

[17] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, La Jolla, California, 18–21 June 1995.

[18] Ninghui Li, Ziqing Mao, and Hong Chen. Usable mandatory integrity protection for operating systems. In *IEEE Symposium on Security and Privacy*, 2007. To appear.

[19] 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.

[20] Andrew C Myers and Babara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering Methodology*, 1999.

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

[22] 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.

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

[24] 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.

[25] P. Saxena, R. Sekar, and Varun Puranik. Efficient fine-grained binary instrumentation with applications to taint tracking. In *ACM/IEEE International Symposium on Code Generation and Optimization*, 2008.

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

[27] R. Sekar. Mcc end user management framework. Technical Report SECLAB06-01, Secure Systems Laboratory, Stony Brook University, June 2006.

[28] 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.

[29] Umesh Shankar, Trent Jaeger, and Reiner Sailer. Toward automated information-flow integrity verification for security-critical applications. In *In Proc. 13th Annual Network and Distributed System Security Symposium (NDSS)*, 2006.

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

[31] Weiqing Sun, Tejas Karandikar, R. Sekar, and Gaurav Poothia. Practical proactive integrity preservation: A basis for malware defense. In *IEEE Symposium on Security and Privacy*, 2008.

[32] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1):77–110, February 2005.

[33] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of 14th Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

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

[35] 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.

[36] 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.

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

[38] Heng Yin, Zhenkai Liang, and Dawn Song. Hookfinder: Identifying and understanding malware hooking behaviors. In *Proceedings of 14th Annual Network and Distributed System Security Symposium (NDSS)*, February 2008.

[39] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of ACM Conference on Computer and Communication Security*, October 2007.

[40] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniquesXFI. In *Symposium on Operating System Design and Implementation (OSDI)*, 2006.