# Static Detection of Unsafe Component Loadings

Taeho Kwon and Zhendong Su

Department of Computer Science, University of California, Davis
{kwon,su}@cs.ucdavis.edu

**Abstract.** Dynamic loading of software components is a commonly used mechanism to achieve better flexibility and modularity in software. For an application's runtime safety, it is important for the application to load only its intended components. However, programming mistakes may lead to failures to load a component, or even worse, to load a malicious component. Recent work has shown that these errors are both prevalent and severe, sometimes leading to remote code execution attacks. The work is based on dynamic analysis by monitoring and analyzing runtime component loadings. Although simple and effective in detecting real errors, it suffers from limited code coverage and may miss important vulnerabilities. Thus, it is desirable to develop effective techniques to detect *all possible* unsafe component loadings.

This paper presents the first *static* binary analysis aiming at detecting all possible loading-related errors. The key challenge is how to scalably and precisely compute what components may be loaded at relevant program locations. Our main insight is that this information is often determined locally from the component loading call sites. This motivates us to design a demand-driven analysis, working backward starting from the relevant call sites. In particular, for a given call site $c$, we first compute its *context-sensitive executable slices*, one for each execution context. Then we emulate the slices to obtain the set of components possibly loaded at $c$. This novel combination of slicing and emulation achieves good scalability and precision by avoiding expensive symbolic analysis. We implemented our technique and evaluated its effectiveness against the existing dynamic technique on nine popular Windows applications. Results show that our tool has better coverage and is precise—it is able to detect many more unsafe loadings. It is also scalable and finishes analyzing all nine applications within minutes.

## 1 Introduction

Dynamic component loading is widely used in software development to build flexible and modular software. Operating systems (OSes) typically provide relevant system calls, such as `dlopen`, to load dynamic components. Once a loading system call is invoked, the underlying OS resolves and loads the specified component. Component resolution depends on how the component is specified—either through the intended component's *full path* or its *file name*. Given a full path, the OS simply uses it for resolution. Given only a file name, the OS searches over a sequence of directories to locate a file with the specified name. Which sequence of directories to search is controlled at runtime by the particular directory search order at the time of system call invocation.

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 122–143, 2012.
© Springer-Verlag Berlin Heidelberg 2012

The flexibility of this common style of component loading does come with a price—it introduces an inherent security concern. For runtime safety and security, an application should only load its intended components. However, as the OS resolves a component only through its name, programming mistakes can lead to the loading of an unintended component with the same name.

Although this issue was known, it is not until recently that studies have shown how prevalent and serious the issue is in practice. In particular, it is shown that unsafe loadings on Microsoft Windows are prevalent and can lead to remote code execution attacks [21]. Remote attacks are possible for two main reasons: 1) the OS looks for a component with a given file name and cannot distinguish malicious ones from benign ones with the same file name; and 2) the default directory search order on Microsoft Windows contains the current directory (*i.e.*, "."), where remote attackers can trick a victim user to download files to via social engineering or by exploiting other vulnerabilities.

Here is an example attack scenario on Windows. An attacker sends a victim user via email an archive that contains an arbitrary `.asx` file and a malicious file named `rapi.dll`. The user extracts the archive file and runs `Winamp 5.58` to open the `.asx` file, the `rapi.dll` is loaded, which leads to a remote code execution attack [21]. Besides archive files, the Carpet-Bomb attack [28] and the WebDAV protocol [2] can be exploited for launching remote attacks. This very issue has also received considerable recent media coverage [11,25,27,36,44]. Microsoft released MS10-087, rated "Critical," to patch Microsoft Office [42]. To mitigate the issue, Microsoft also released a fix-it tool to control the directory search order by introducing a new registry key [17, 26]. However, it changes the default system-wide setting and leads to backward compatibility issues. Fundamentally, this is a safe programming issue. Microsoft provides programming guidelines for safe dynamic loading [10] and is conducting an ongoing investigation to secure the loading procedure [23].

As the root cause of the issue is unsafe programming, it is important to detect this class of dangerous programming mistakes. Kwon and Su [21] proposed a dynamic technique to detect unsafe component loadings. This technique collects at runtime loading-related information—such as the target component to be loaded, the directory search order, and the actually loaded component—at each of the invocation sites for the loading system call. It then performs an offline analysis to detect two types of unsafe loadings: *resolution failure* and *unsafe resolution*. A resolution failure happens when the target component is not found, while an unsafe resolution happens when other directories are searched before the directory where the loaded component resides. Besides crashing an application, unsafe loadings also make the application vulnerable to component hijacking.

Although the proposed dynamic technique [21] is effective at detecting real unsafe loadings, it may miss errors because of limited code coverage, an inherent weakness of dynamic analysis. We illustrate this issue using *delayed loading*, an optimization to postpone the loading of infrequently used components until their first use. Delayed loading is challenging for dynamic detection because it is difficult to trigger all delayed loadings at runtime. Figure 1 shows a code snippet that uses delayed loading in Microsoft Windows. The code shows two functions `f1` and `f2` that use components registered for delayed loading. In particular, `f1` and `f2` retrieve the addresses of `OpenPrinter` exported by `WINSPOOL.DRV` and `GetSaveFile` exported by `COMDLG32.DLL` respectively.

Although the example only shows two functions `f1` and `f2`, in practice, there are often many more. The infrequent use of the components makes it difficult, if not impossible, to trigger all possible loadings at runtime. Although we have illustrated the problem using delayed loading, poor coverage of dynamic analysis is a general concern for detecting unsafe loadings, as our results also confirm (*cf.* Section 3).

In this paper, we present the first *static* analysis to detect unsafe loadings from program binaries. Two pieces of essential information are needed: 1) all components that may be loaded at each loading call site, and 2) the safety of each possible loading. While the second part is straightforward, the key challenge lies in the first part—how to precisely and scalably compute the possible loadings. Our *key observation* is: for a given invocation of the loading system call, the set of possible loaded components is determined by the system call's parameter values, which are often determined through computations that originate not far from the call site. From these observations, we design a two-phase analysis: *extraction* and *checking*. The extraction phase is *demand-driven*, working backward from each loading call site to compute the set of possible loadings; the checking phase determines the safety of a loading by examining the relevant directory search order at the call site.

**Context-Sensitive Emulation.** To realize the backward computation of parameter values during the extraction phase, we introduce *context-sensitive emulation*, a novel combination of slicing and emulation. For a given call site, we extract its context-sensitive *executable* slices w.r.t. its parameters, one for each execution context. We then emulate the slices to compute the parameter values.

*Incremental and Modular Slicing.* One technical obstacle is how to compute backward slices scalably. Standard slicing techniques [1, 5, 13, 30, 35, 38] are based on computing a program's complete system dependence graph (SDG) *a priori* and are thus limited in scalability. Because we only need to consider loading call sites and the execution paths to compute the parameter values to the calls are usually relatively short, only a small fraction of the complete SDG is relevant for our analysis. This motivates the use of an *incremental* and *modular* slicing algorithm (*cf.* Section 2)—incremental because we build the slices lazily when necessary; modular because when we encounter a function call `foo(x,y)`, we use an inferred summary of what dependencies `foo`'s parameters and return value have in analyzing the caller. At the end, we connect the function-level slices in the standard way by linking formal and actual parameters.

```
1  void f1() {
2      ...
3      pDelayDesc1 = &WINSPOOL_DRV_DelayDesc;
4      // WINSPOOL_DRV_DelayDesc.dllname = "WINSPOOL.DRV"
5      func_addr = __delayLoadHelper2(
6          pDelayDesc1, "OpenPrinter"
7      );
8      ...
9  }
10 void f2() {
11     ...
12     pDelayDesc2 = &COMDLG32_DLL_DelayDesc;
13     // COMDLG32_DLL_DelayDesc.dllname = "COMDLG32.DRV"
14     func_addr = __delayLoadHelper2(
15         pDelayDesc2, "GetSaveFile"
16     );
17     ...
18 }
19 int __delayLoadHelper2(pImgDelayDesc, funcName) {
20     hMod = pImgDelayDesc->hMod; // init value = 0
21     if (hMod == 0) {
22         target_dllname = pImgDelayDesc->dllname;
23         hMod = LoadLibrary(target_dllname);
24         pImgDelayDesc->hMod = hMod;
25     }
26     func_addr = GetProcAddress(hMod, funcName);
27     return func_addr;
28 }
```

**Fig. 1.** Motivating example

*Emulation of Context-sensitive Slices.* Once we have computed the backward slice $s$ w.r.t. a given loading call site, we need to compute possible values for the relevant parameters. One natural solution is to perform standard symbolic analysis on the slice to compute the values. The main challenge for this approach is the difficulty in reasoning symbolically about system calls because the relevant parameters often depend on complex, low-level system calls. For example, many Windows applications invoke the system call `RegQueryValueExW` to retrieve the fullpath of the target specification stored in the registry key. The system call invokes more than 100 distinct system calls exported by five libraries. To symbolically analyze the system call, it is necessary to symbolically execute its invoked system calls as well, leading to path explosion. Thus, it is difficult in practice to engineer and scale symbolic analysis to compute the possible values of the parameters.

To overcome this difficulty, we use emulation. In particular, we generate, from the backward slice $s$, a set of *context-sensitive executable sub-slices*, which we then *emulate* to compute the parameter values (*cf.* Section 2). Essentially, we inline callees' function-level slices in each execution context to produce $s$'s sub-slices $s_1, \ldots, s_n$. Instructions in each sub-slice $s_i$ are next emulated topologically, respecting their data- and control-flow dependencies.

For evaluation, we implemented our technique in a prototype tool for Windows applications. We evaluated our tool's effectiveness against the previous dynamic tool [21] in terms of precision, scalability, and coverage. Results on nine popular applications show that our tool is precise and scalable (*cf.* Section 3). For example, it took less than two minutes to analyze each of the nine test subjects, including large applications such as Acrobat Reader, Quicktime, and Safari. The results also show that our proposed context-sensitive emulation achieves orders of magnitude reduction in the size of the code needed to be analyzed and crucially contributes to the scalability of our technique. In terms of coverage, our tool detected many more possible unsafe loadings and nicely complements the dynamic technique.

**Main Contributions**

- We have developed the first static binary analysis to detect unsafe component loadings. Because of its scalability and higher code coverage, our technique effectively complements the existing dynamic technique.
- We have proposed context-sensitive emulation, an effective approach that combines slicing and emulation for the precise and scalable analysis of runtime values of program variables.
- We have implemented our technique and evaluated its effectiveness by detecting unsafe loadings in nine popular Windows applications.

The rest of this paper is organized as follows. Section 2 presents a detailed description of our static detection algorithm. We describe our implementation and evaluation in Section 3. Finally, Section 4 surveys additional related work, and Section 5 concludes with a discussion of future work.

## 2 Static Detection Algorithm

In this section, we present background information on unsafe component loadings and details of our analysis.

### 2.1 Background

Dynamic component loading is commonly supported by operating systems through specific system calls that take as input a full path or file name for the intended component. For example, Microsoft Windows provides component-loading system calls such as `LoadLibraryA`. Once such a system call is invoked, the OS resolves the target component as follows:

- The target component can be specified by its full path or its file name.
- When the full path is used, the OS directly resolves the target using the provided full path.
- Otherwise, if file name is used and is known by the OS, the full path of the specified file is predefined. For example, `KERNEL32.DLL` is known by Microsoft Windows and its full path is predefined as `"C:\WINDOWS\SYSTEM32\KERNEL32.DLL"`.
- If the given file name is unknown to the OS, it iterates through the predefined search directories to locate the first file with the specified file name.

To formalize the component resolution process, it is necessary to model the *file system state*, because even the same component-loading code may result in different resolutions under different file system states. We define a file system state $s$ to be the set of full paths of all files stored on the current file system.

**Definition 21 (Component Resolution).** *A component resolution function $R$ takes a component specification $f \in \Sigma^*$, a directory search order $d = \langle d_1, \ldots, d_n \rangle \in \Sigma^* \times \ldots \times \Sigma^*$ and a file system state $s$, and returns a resolved full path $\pi \in \Sigma^*$, where $\Sigma$ denotes the alphabet used to specify files and directories.*

- *If $f$ is a full path,*

$$R(f, d, s) = \begin{cases} f \text{ if } f \in s; \\ \epsilon \text{ otherwise.} \end{cases}$$

  *where $\epsilon$ is the empty string.*
- *If $f$ is a file name,*

$$R(f, d, s) = \begin{cases} \pi & \text{if } f \text{ is known to the OS as } \pi; \\ d_k + \backslash + f & \text{if } S = \{i \mid d_i + \backslash + f \in s\} \\ & \quad \wedge\, S \neq \emptyset \ \wedge\ k = \mathsf{min}(S); \\ \epsilon & \text{otherwise.} \end{cases}$$

  *where "+" denotes string concatenation.*

We next formalize component loading, for which we need to consider the currently loaded components. The reason is that the OS does not load the same component multiple times. In our formalization, we let $\Pi$ denote the set of full paths of all the currently loaded components.

**Definition 22 (Component Loading).** *Given the loaded components $\Pi$, a component loading function $L$ takes a component specification $f \in \Sigma^*$, a directory search order $d = \langle d_1, \ldots, d_n \rangle \in \Sigma^* \times \ldots \Sigma^*$, a file system state $s$, and the set of loaded components $\Pi$, and returns a resolution success or failure:*

$$L(f, d, s, \Pi) = \begin{cases} \textit{success} & \text{if } R(f, d, s) \notin \{\epsilon\} \cup \Pi; \\ \textit{failure} & \text{otherwise.} \end{cases}$$

The formalized component loading mechanism in Definition 22 is commonly used on major operating systems. However, as the OS determines a target component only through its name, unsafe programming can make software load an unintended component with the same name. Attackers can exploit this security vulnerability by modifying the file system state. In particular, the loading of a target component can be hijacked if a malicious file with the same name can be created in a directory searched before the directory where the intended component resides. This component hijacking can be misused for local or remote attacks [21].

To formalize unsafe component loading, it is necessary to determine the current file system state as whether or not a component loading is safe is relative to a file system state. We first define a *normal file system state* w.r.t. an application $p$.

**Definition 23 (Normal File System State).** *A file system state $s$ is* normal *w.r.t. an application $p$ if no unintended components are loaded while $p$ executes in state $s$. We use $s_p$ to denote a normal file system state w.r.t. the application $p$.*

We formalize two types of unsafe loadings: *resolution failure* and *unsafe resolution*. We use $R_p$ and $L_p$ to denote component resolution and component loading performed by an application $p$, respectively.

**Definition 24 (Resolution Failure).** *For an application $p$, a resolution failure occurs at runtime if $R_p(f, d, s_p) = \epsilon$. In this case, with a full path specification $f$, an arbitrary file with the same full path $f$ can hijack the component loading. If $f$ is file name, an attacker can hijack this loading by placing a file (or tricking the user to place a file) with the specified name $f$ in any directory $d_i$* writable *by the attacker under the search order $d = \langle d_1, \ldots, d_n \rangle$.*

**Definition 25 (Unsafe Resolution).** *For an application $p$, an unsafe resolution occurs at runtime if the following conditions hold: 1) $f$ is the file name of the target component and unknown to the OS; 2) $R_p(f, d, s_p) = d_k + \backslash + f \wedge k > 1$; and 3) $L_p(f, d, s_s, \Pi) = success$. In this case, an attacker can hijack the loading by placing a file (or tricking the user to place a file) with the specified name $f$ in any* writable *directory $d_i$ by the attacker where $i < k$.*

To avoid unsafe loadings, it is necessary for developers to specify the target component in a safe manner. We define safe target component specifications as follows.

**Definition 26 (Safe Component Spec).** *Under a given threat model, a loading specification for an application $p$ is safe if either of the following holds: 1) if $f$ is a full path, $R(f, d, s_p) \neq \epsilon$ and the attacker cannot overwite $f$ or trick the user to overwrite $f$; and 2) if $f$ is an unknown file name to the OS, $R(f, d, s_p) = d_i + \backslash + f$ and the attacker cannot place a file or trick the user to place a file named $f$ in any of the $d_j$ for $1 \leq j \leq i$.*

If a loading specification is unsafe, it leads to resolution failure or unsafe resolution. While the first condition checks the resolution failure for the fullpath specification, the second condition checks whether the filename specification leads to resolution failure or unsafe resolution. As many Windows users have the administrator privilege, a realistic

threat model under Windows is that an attacker may be able to trick a (non-malicious) user to place a malicious component in a desired directory to hijack component loading. We have adopted this threat model in our evaluation.

### 2.2 Detailed Analysis

We now present the details of our analysis. Our technique statically detects unsafe component loadings to achieve high coverage. It first extracts the target component specifications from possible code region executed at runtime and checks their safety based on Definition 26.

**Program**

```
1 int main() {
2  x = rand()%2;
3  if (x == 0) {
4   A = LoadLibrary("A");
5   A.foo1();
6  }
7  else {
8   B = LoadLibrary("B");
9   B.bar1();
10 }
11 }
```

**Component A**

```
1 void foo1() {
2  C = LoadLibrary("C");
3  C.foo2();
4 }
```

**Component C**

```
1 void foo2() {
2 ...
3 }
```

**Component B**

```
1 void bar1() {
2  D = LoadLibrary("D");
3  D.bar2();
4 }
```

**Component D**

```
1 void bar2() {
2 ...
3 }
```

**Fig. 2.** Component-interoperating code

The executed code region is determined by loaded components. Figure 2 depicts the component loading code whose execution path is controlled by a random variable `x`. If `x` is zero, `foo1` of component `A` and `foo2` of component `C` are executed. Otherwise, `bar1` of component `B` and `bar2` of component `D` are executed. Our observation is that each execution path covers the partial code region of the loaded components. For example, if `x` is zero, the partial code regions of components `Program`, `A`, and `C` are executed. From these observations, we design our static detection as shown in Figure 3: *extraction* and *checking*. From the extraction phase, we obtain a set of the target component specifications from the components that can be loaded at runtime. In the checking phase, we evaluate the safety of each target specification based on Definition 26.

**Extraction Phase.** A component can load other components at loadtime or runtime. This loading introduces *loadtime* and *runtime* dependencies among components [41]. Based on these dependencies, we determine components that can be loaded during program execution. Specifically, we recursively resolve the components from the program file based on their loadtime and runtime dependencies. To resolve the dependent components, the corresponding target specifications, *i.e.*, full path or file name, are needed. For loadtime dependencies, compilers specify the dependent components in the executable format. For example, the names of the loadtime dependent components are stored in `IMAGE_IMPORT_DIRECTORY` with the PE format [24]. To obtain the specifications of the runtime dependent components, we compute values of parameters to component-loading system calls. This suffices for our setting because the program dynamically loads components via the system calls and their parameters determine the loaded components.

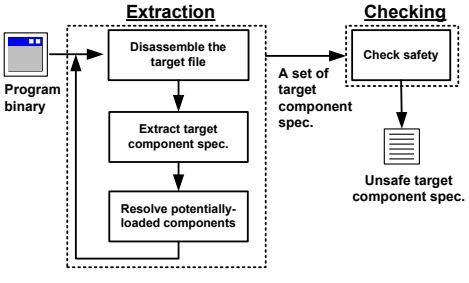



**Fig. 3.** Detection framework

As an example of recursive resolution, we search the components that can be loaded by `Program` in Figure 2. Suppose that components `E` and `F`, which have no loadtime

```
1  PUSH  EAX
2  PUSH  EAX
3  PUSH  offset 0x7D61AC5C; "xpsp2res.dll"
4  CALL  DWORD PTR DS:[LoadLibraryExA]
```

(a) Memory indirect

```
1  MOV   EBX, DWORD PTR DS:[LoadLibraryW]
2  PUSH  offset 0x65015728; "CABINET.DLL"
3  CALL  EBX
```

(b) Register indirect

**Fig. 4.** Two types of component-loading call sites

or runtime dependent components, implement the `rand` and `LoadLibrary` functions, respectively. In this case, `Program` loads components `E` and `F` on its startup. Regarding runtime dependencies, `Program` dynamically loads components with the specifications, `"A"` on line 4 and `"B"` on line 8. From this information, we can detect the potentially-loaded components by simulating component resolution. Similarly, we can infer that `C`, `D` and `F`, which are loaded by `A` and `B`. Because `C` and `D` have no loadtime and runtime dependent components, we stop the resolution process. Thus, we detect the seven components potentially loaded at runtime: `Program`, `A`, `B`, `C`, `D`, `E`, and `F`.

The key step of the extraction phase is to obtain the target specification for component loading in a binary. The specification of a loadtime dependent component can be easily obtained from the binary file format. However, extracting the specification of a runtime dependent component is nontrivial because it often requires to locate the code relevant to the value of the specification and analyze its execution. For example, the target component specification for system libraries under Microsoft Windows is sometimes determined by concatenating the system directory path and the file name. To obtain the specification, it is necessary to extract the related code and analyze its execution result.

The concrete value of the parameter to the component-loading system call serves as the specification for the runtime dependent component. From this observation, we extract the specification by searching for the program variable for the specification and then computing its value via *context-sensitive emulation*, a novel combination of backward slicing and emulation. We describe details of the extraction in the following sections.

**Searching Program Variable for Specification.** In binary code, invoking the component-loading system calls follow the `stdcall` calling convention [45]. When parameters are passed to the call site, they are pushed from right to left. For example, Figure 4(a) represents the binary code corresponding to `LoadLibraryExA(0x7D61AC5C, EAX, EAX)`. Based on the parameter passing mechanism, we locate the program variable, *e.g.*, a register or a memory chunk, which stores the target specification. In particular, we detect the call site for component loading via static taint data analysis and then extract the input operands of the instructions passing the parameter to the call site. We describe details of each step in the rest of this section.

***Locating Component-loading Call Sites.*** In this phase, we aim at finding the call site for component loading in a binary. Our observation is that software stores the address of the system call implementation in its memory space and utilizes it in the call sites for component loading at runtime. Figure 4 shows the two types of component-loading call sites in a binary, which are *memory indirect* and *register indirect*. The main difference between them is what type of program variable stores the address of the

component-loading system call at the call site. While the memory indirect type stores the address in a memory chunk, the register indirect type stores the address in a register, *e.g.*, line 4 in Figure 4(a) and line 3 in Figure 4(b).

Based on this observation, we locate the component-loading call sites through static taint data analysis. In particular, we define the taint sources and the taint sinks as follows:

- *Taint source*: an instruction that references a memory chunk that stores the address of the component-loading system call.
- *Taint sink*: a branch instruction, *e.g.*, `call`, whose target address is tainted. We consider the taint sink instructions as the call sites.

We now present examples on how to detect call sites. In Figure 4(a), line 4 serves as not only the taint source but also the taint sink, *i.e.*, the component-loading call site, because it is the branch instruction, accessing a memory chunk that stores the address of `LoadLibraryExA`. For Figure 4(b), line 1 is the taint source, accessing the address of the `LoadLibraryA`, and line 3 is the taint sink, because it is the `call` instruction whose target is the address, stored in `EBX`.

***Extracting Parameter Variables.*** Once a call site is located, we extract the program variables for the target specification from the predefined number of the instructions to pass the parameters to the call site. In particular, we detect the instructions, *e.g.*, `PUSH`, to initialize the top of stack backward from the call site. Because the number of parameters of a component-loading system call is known, we can precisely extract all the variables to define this target specification. For example, the call site in Figure 4(a) invokes `LoadLibraryExA`, and it has three parameters, *i.e.*, `0x7D61AC5C`, `EAX`, and `EAX`, via the instructions on lines 1–3.

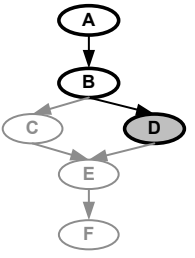
**Context-Sensitive Emulation.** In this phase, we compute the concrete values of the parameter variables extracted in Section 2.2. The computation may seem trivial at first. For example, the memory chunk at `0x7D61AC5C` in Figure 4(a) contains the target specification, `"xpsp2res.dll"`. However, the computation is in fact challenging because it is necessary to extract the code to compute the variable, requiring interprocedural data flow analyses (*cf.* Figure 1). Also, we need the runtime information of the code to obtain the concrete values of the variable. Symbolic analysis can serve as a potential solution. However, as we mentioned in Section 1, symbolic analysis suffers from poor scalability and is limited in handling system calls, which are often complex.

To address this problem, we introduce *context-sensitive emulation*, which novely combines backward slicing and emulation. Based on this combination, we can scalably and precisely compute the values of the variables of interest. We describe its details in the rest of this section.

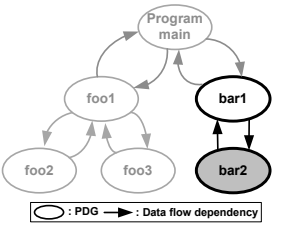
***Backward Slicing.*** This phase performs the interprocedural backward slicing w.r.t. the parameter variable, extracting the instructions to compute the variable. This problem has been extensively studied, and many slicing algorithms [1,5,13,30,35,38] have been proposed. These algorithms commonly solve the graph reachability problem over a System Dependence Graph (SDG) [13], a set of Program Dependence Graphs (PDGs) [12]

and edges capturing data flow dependencies among them. In particular, a SDG is constructed beforehand based on an exhaustive data flow analysis over the subject program. Then, the slicing outcome is determined by traversing the SDG from the given slicing criteria. Although the approach has been widely used, it is not appropriate for our problem setting. The reason is that binary files are generally composed of a large number of instructions, and an exhaustive data flow analysis over all the instructions is very expensive, leading to limited scalability.

Our key observation is that the parameter values are often locally determined, that is the execution paths to compute the variables are relatively short. Thus, exhaustive data flow analysis is not be necessary to extract backward slices w.r.t. the given slicing criteria. Figure 5 shows examples of the unnecessary data flow analysis during intraprocedural and interprocedural backward slicing.




(a) Intra-backward




(b) Inter-backward

**Fig. 5.** Unnecessary data flow analysis

Figure 5(a) shows an example of the CFG for constructing the PDG. Suppose that we perform intraprocedural backward slicing w.r.t. the instruction `D`. In this case, the bold instructions often only affect the instruction `D` in terms of control flow. It is possible that the instruction `D` can be affected by the instructions without control flow dependencies. For example, the instruction `E` initializes a variable and the instruction `B` reads it. However, this case rarely happens in our problem setting in practice, because the parameters for the specification are generally computed by the instructions executed before the component-loading call sites.

Suppose that Figure 5(b) depicts the SDG for the interprocedural backward slicing. If the instructions of the bold PDGs for `bar1` and `bar2` are only traversed during slicing, it is not necessary to perform data flow analysis on the instructions of the grayed PDGs. Because the SDG consists of a large number of PDGs in binary and the target specifications are often locally determined, most of the PDGs are not relevant for interprocedural backward slicing w.r.t. the parameter variables for the target specifications.

Based on this insight, we design our slicing technique as demand-driven, reducing the unnecessary analysis of data flow dependencies. In particular, we perform interprocedural backward slicing by incrementally combining intraprocedural backward slices whose slicing criteria are determined when necessary.

*Intraprocedural backward Slicing.* For each intraprocedural backward slicing, we analyze only the data flow dependencies among the instructions that are control dependent on the given slicing criteria. To this end, we construct the PDG based on the *predecessor subgraph* w.r.t. the slicing criterion under the CFG. Thus, we can avoid the analysis of the data flow dependency among the instructions not traversed during slicing. Suppose that we perform intraprocedural backward slicing w.r.t. the instruction `D` in the CFG shown in Figure 5(a). If we construct the PDG based on the CFG, the data flow

dependencies among all the instructions in the CFG are analyzed. However, the grayed instructions do not affect the instruction `D` in terms of control flow dependencies. By constructing the PDG based on the subgraph composed of the bold instructions, *i.e.*, the predecessor subgraph w.r.t. the instruction `D`, we can avoid some unnecessary data flow analysis when performing slicing.

One challenge for PDG construction is caused by the call site instructions. Because functions are not generally monolithic, it is necessary to identify which call sites affect the slicing criteria. Although traversing the SDG provides such information, it requires the computation of significant amount of unnecessary data-flow dependencies (*cf.* Figure 5(b)). To address this problem, we utilize the prototypes of the functions invoked at the call sites. Specifically, we consider a call site instruction as a non-branching instruction during our PDG construction, and analyze the data flow dependencies related to the call site in terms of the prototype of the callee function. For example, a call site invokes a function `foo` whose prototype is `int foo(in,inout)`. In this case, the call to `foo` is considered to be an instruction that uses the first and second parameters and defines the second parameter and the return variable. Based on this information, we can effectively determine the data flow dependencies between the call site instructions and the slicing criteria without a whole SDG traversal.

*Interprocedural backward Slicing.* As aforementioned, an exhaustive SDG construction often leads to significant amount of data flow analysis that is unnecessary for interprocedural backward slicing. To address this problem, we construct the interprocedural backward slices incrementally combining the intraprocedural backward slices whose slicing criteria are chosen in a demand-driven manner.

There are two key challenges for this demand-driven combination. First, it is necessary to determine the new slicing criteria if the interprocedural backward slice consists of multiple intraprocedural backward slices. For example, we construct the interprocedural backward slice in Figure 5(b) by combining the two intra-backward slices extracted from functions `bar1` and `bar2`. In this case, we need to determine the new slicing criteria in the `bar1` function. Second, the composed interprocedural backward slice needs to be easily handled for the later emulation phase.

Our basic idea for building the new slicing criteria is that the interprocedural data flow dependencies are captured by parameter passing. In SDG-based slicing, the PDGs are connected using the edges that model parameter passing, which are traversed to analyze the dependencies. Based on this idea, we choose the slicing criteria as follows. Suppose that an intraprocedural backward slice $s$ is extracted from an instruction whose input operand is initialized through parameter $p$ of the function $f$. In this case, we determine the new slicing criterion as the parameter

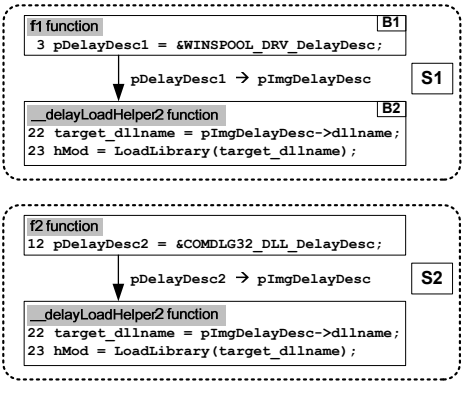



**Fig. 6.** Example context-sensitive backward slices

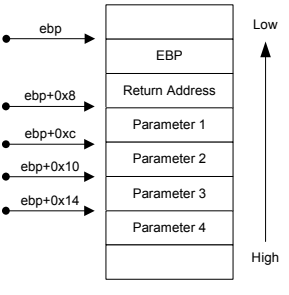
variable corresponding to the parameter $p$. To locate this parameter variable, we use *caller-callee relationship* and the *callee's function prototype*. In particular, we detect the call site for function $f$ and analyze $f$'s function prototype to obtain the index of the parameter corresponding to $p$. For example, the intraprocedural backward slice w.r.t. the `target_dllname` in Figure 1 uses the first parameter, *i.e.*, `pImgDelayDesc`, of `__delayLoadHelper2`. As two call sites on lines 5–7 and lines 14–16 invoke `__delayLoadHelper2`, we choose their first parameter variables, *i.e.*, `pDelayDesc1` on line 6 and `pDelayDesc2` on line 15, as the new slicing criterion.

Once the new slicing criterion is determined, we construct the interprocedural backward slice by composing the intraprocedural backward slices and use the composed slice in the emulation phase. One simple method for composing the intraprocedural slices is to collect the instructions of each intraprocedural backward slice. For example, the interprocedural backward slice w.r.t. the `target_dllname` in Figure 1 consists of the instructions of three intraprocedural backward slices w.r.t. the slicing criteria, *i.e.*, `target_dllname`, `pDelayDesc1`, and `pDelayDesc2`. However, this simple method produces *context-insensitive* slices, making the emulation phase complex. In particular, when emulating each instruction of the context-insensitive slice, we have to assume that the values of its operands are determined under all of its calling contexts.

To better support emulation, we combine the intraprocedural backward slices to construct a set of *context-sensitive interprocedural backward slices*. In particular, for a given intraprocedural backward slice $s$, if multiple new slicing criteria, $p_1 \dots p_n$, are determined, the set of the context-sensitive slices are constructed as $\{s_i \cup s | s_i = \cup_{p_i}$ intraprocedural backward slice w.r.t. $p_i$ where $1 \leq i \leq n\}$. Thus, we can more straightforwardly use the context-sensitive slices to compute possible concrete values of the target component specification. For example, Figure 6 shows the context-sensitive interprocedural backward slices w.r.t. `target_dllname` in Figure 6, We can compute the possible values of `target_dllname` by emulating them. We describe more details of our backward slicing phase in an earlier version of this paper [22].

```
foo:
1  ...
2  mov eax, [ebp+0xc] ; 2nd
3  ...
4  mov [ebp+0x8], eax ; 1st
5  ...
6  mov eax, [ebp+0x10] ; 3rd
7  ...
8  mov [ebp+0x14], eax ; 4th
9  ...
```

(a) Parameter access




(b) Stack layout

**Fig. 7.** Function prototype analysis

*Function Prototype Analysis.* The backward slicing phase relies on function prototypes, but such information is often unavailable in binary code. Our solution to this problem is as follows. For a given function $f$, its parameters are stored in fixed locations during $f$'s execution. Thus, we infer its prototype by analyzing how the instructions of the function access the memory chunks for the parameters, *i.e.*, read or write.

Figure 7 shows an example of our proposed prototype analysis for the `foo` function. Suppose that Figures 7(a) and 7(b) show part of `foo` and the stack layout at the beginning of the function's execution, respectively. In this case, the `idx`-th parameter is stored at the address $\texttt{ebp} + 4 \times (\texttt{idx} + 1)$ where the stack is aligned by four bytes. From this observation, we can infer `foo`'s prototype. It reads data from the memory chunks for its second and third parameters, and initializes the memory chunks for its first and fourth parameters, *i.e.*, its function prototype is `"eax foo(inout,in,in,inout)"`. Here we assume that its result is returned through the `eax` register.

To improve the precision of our prototype inference, we use the following effective heuristic. If the effective address of the memory chunk, obtained by the `lea` instruction, is passed to the function, we consider it as the `inout` parameter. The effective address corresponds to a pointer variable and the memory chunk that it points to is often initialized during function execution. Although this heuristics may increase the size of the computed slice, it is sufficient to compute possible values of the slicing criteria via emulation.

***Emulation Phase.*** In this phase, we compute the possible values of the target component specification by emulating its corresponding context-sensitive slices. There are three challenges for slice emulation. The first challenge is how to schedule the instructions because we do not know their runtime execution sequence. If the instructions are incorrectly scheduled, they may violate the data and control flow dependencies among them, which may lead to imprecise results or emulation failures. The second challenge is how to pass function parameters. Although parameter passing captures useful data flow dependencies, the context-sensitive slices do not explicitly specify the dependencies. The third challenge is how to handle the call site instructions. Because we perform the data flow analysis by considering a call site as an instruction, the backward slice does not contain detailed code of the callee function.

*Scheduling Algorithm.* To develop a practical scheduling algorithm, we have analyzed all 682 backward slices extracted from nine popular Windows applications (*cf.*, Table 1). We have observed that all the extracted slices form directed acyclic graphs. Therefore, we schedule the basic blocks in their topological order w.r.t. dataflow dependency. We then determine the order of the instructions of each basic block w.r.t. their sequence in the original code. For example, Figure 8 shows the data flow dependency among the basic blocks of the first slice in Figure 6. In this case, we schedule the basic blocks as `B1`, `B2`, and `B3`. For each basic block, the sequence of its instructions is determined as

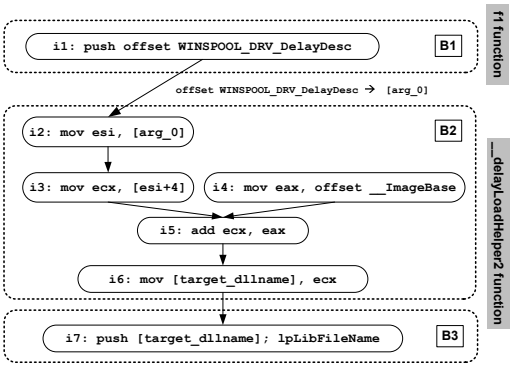



**Fig. 8.** Data-flow dependency among basic blocks

follows: i1, i4,i2,i3,i5,i6, and i7. The scheduled sequence of the instructions does not violate the data- and control-flow dependency among them.

*Parameter Passing.* To handle parameter passing, we initialize the stack frame before emulating the callee function. In particular, suppose that a parameter $p$ is passed to a function $f$. In this case, before emulating $f$'s basic blocks, we reserve the stack frame and initialize its memory chunk for the parameter with the concrete value of $p$. The location of the memory chunk is determined by the index of the passed parameter. For example, the address of the memory chunk for the idx-th parameter can be computed by ebp $+ 4 \times$ (idx $+ 1$), (*cf.* Figure 7).

For example, Figure 8 shows how we handle the parameter passing from f1 to __delayLoadHelper2. When B1 is emulated, top of the call stack for f1 stores offset WINSPOOL_DRV_DelayDesc. Assuming that the initial value of esp for emulating B2 is equal to 0x13f258, the stored value initializes a memory chunk at arg_0=0x13f258+4×2, because it corresponds to the first parameter to __delayLoadHelper2. The instructions use arg_0 to reference the first parameter (*e.g.*, i2).

*Call Site Instruction.* To obtain the possible values of the target component specification, it is necessary to emulate the call site instruction. If the code of the invoked function resides in the current file, we can simply emulate the corresponding code. However, if the call site invokes a system call, we may not be able to obtain the code from the current file. Figure 9 shows an example slice with external library calls where each edge represents data flow dependency between two instructions. The slice determines the fullpath of the target component by concatenating the path to the system directory with a string \kernel32.dll. In this case, the instructions invoked by i5 and i10 are not available in the current file. In particular, GetSystemDirectoryW and wcscat_s are implemented in KERNEL32.DLL and MSVCRT.DLL, respectively.

One natural solution is to perform instruction-level emulation over the system call implementations obtained from the corresponding libraries. However, this is not practical because system call implementations typically have a large number of instructions and lead to poor scalability.

Thus, we do not emulate the system call code at the instruction-level. Instead, we use code to model the side effects of system calls and execute the models. For example, Figure 10(a) and Figure 10(b) show the stack layout before and after processing i5 shown in Figure 9. The example models the side effect of GetSystemDirectoryW: 1) retrieve the two parameters from the stack; 2)

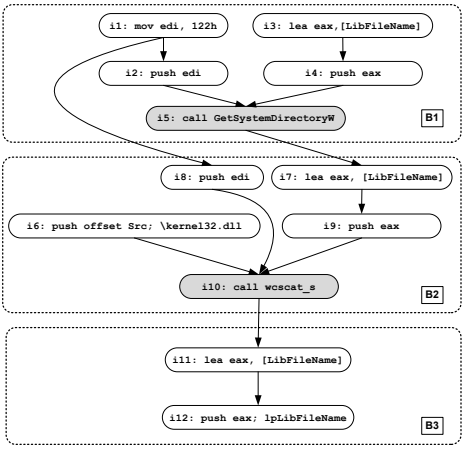



**Fig. 9.** Backward slice with external library calls

obtain the system directory path by invoking `GetSystemDirectoryW`; 3) write the directory path to the memory chunk pointed to by the first parameter; 4) copy the system call's return value to `eax` register and adjust the `esp` register to clean up the stack frame.

Based on the technique discussed above, we can emulate the context-sensitive slices to compute the possible values of the target component specification. For example, we can compute the value, `"C:\Windows\System32\KERNEL32.DLL"`, of `lpLibFileName` by emulating the backward slice in Figure 9.

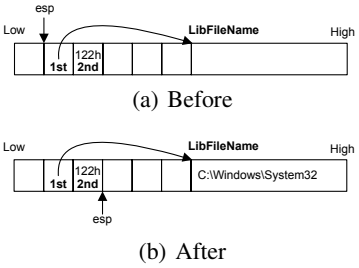



**Fig. 10.** Side effects of `i5` in Figure 9

**Checking Phase.** In this phase, we evaluate the safety of the target component specifications obtained from the extraction phase. To this end, for each specification, we check whether or not the safety conditions in Definition 26 are satisfied. In particular, when the fullpath is specified, we check whether or not the specified file exists in the normal file system. For the filename specification, we consider that a specification can lead to unsafe loading if the target component is unknown and the OS cannot resolve it in the directory that is first searched on the normal file system. Note that the names of the known component and the first directory searched by the OS for the resolution are predefined [8, 9, 21].

As an example of this phase, we check the component loading discussed in the attack scenario in Section 1. When opening the `.asx` file, `Winamp 5.58` tries to load `rapi.dll`. In this case, OS iterates through a list of predefined directories [9] to locate the file named `rapi.dll`. However, no such file is found during the iteration. Thus, this loading is unsafe, because attackers can hijack this loading by placing malicious `rapi.dll` files in the checked directories. In particular, the current working directory, one of the directories, is determined as the same directory as the `.asx` file, leading to the remote code execution attack. Suppose that the file named `rapi.dll` exists in the directory first searched, *i.e.*, the `Winamp` program directory. In this case, this loading is safe, because there is no directory such that attackers can misuse for hijacking.

## 3 Empirical Evaluation

In this section, we evaluate our static technique in terms of precision, scalability, and code coverage. We show that our technique scales to large real-world applications and is precise. It also has good coverage, substantially better than the existing dynamic approach [21].

### 3.1 Implementation

We implemented our technique on Windows XP SP3 as a plugin to IDA Pro [15], a state-of-the-art commercial binary disassembler. Our IDA Pro plugin is implemented using IDAPython [16] and three libraries: 1) NetworkX [29] for graph analysis, 2) PyEmu [32] for emulation, and 3) pefile [31] for PE format analysis.

For the precise analysis of binaries, it is important to map between C-like variables and memory regions accessed by instructions. We adapt the concept of an *abstract*

*location* (*a-loc*) [3], which models a concrete memory address in terms of the base address for a memory region and a relative offset. For example, the a-loc for `&a[4]` is `mem_4` where `mem` is the base address of the array `a` and 4 is the relative offset from the base address. Refer to Balakrishnan and Reps [3] for more details.

Backward slicing in our technique requires function prototypes of system calls. To this end, we analyzed the files in the system directory and collected prototypes for 3,291 system calls.

To emulate the code modeling side effects of system calls, we need to determine what system call is invoked through a given call site instruction. We have extended PyEmu's `set_library_handler` function so that it can register callback functions for external function calls. We implemented the callbacks for 68 system calls used by the extracted slices.

To implement our tool, it is necessary to extract CFGs and call graphs from binaries. We leverage the disassemble result of IDA Pro in our current implementation. It is well-known that indirect jumps can be difficult to resolve for binaries. Although IDA Pro does resolve certain indirect jumps, it may miss control-flow and call dependencies, which is one source of incompleteness in our implementation.

### 3.2 Evaluation Setup and Results

We aim at detecting unsafe component loadings in applications. Because the detection of unsafe loadings from the system libraries is performed by the operating system, we only resolve the application components in the extraction phase.

The checking phase for a target specification requires the information on the first directory searched by the OS for the resolution and the relevant normal file system state (*cf.*, Definition 26 and Section 2.2). We obtain this information by analyzing the extracted parameters and the applications. For example, suppose that an application $p$ loads an unknown component by invoking `LoadLibrary` with the component's file-name. In this case, we can infer the directory where $p$ is installed because Microsoft Windows first checks the directory where $p$ is loaded. Regarding the normal file system state, we installed the applications with the default OS configuration and detected unsafe loadings for each application. In this setting, we assume that 1) the default file system state is normal, and 2) the application itself is benign in that does not cause installed applications to have unintended component loadings.

**Detection Results and Scalability.** Table 1 shows our analysis results on nine popular Windows applications. We chose these applications as our test subjects because they are important applications in wide-spread use. The results show that our technique can effectively detect, from program binaries, unsafe component loadings potentially loaded at runtime. Note that the results of the extraction phase for `Seamonkey` and `Thunderbird` are identical. This is likely because both applications are part of the Mozilla project and use the same set of program components.

We rely on IDA Pro for disassembling binaries, and Table 1 includes the time that it took IDA Pro to disassemble the nine applications. This time dominates our analysis time as we show later. These are large applications, and also we only need to disassemble the code once for all the subsequent analysis.

**Table 1.** Analysis of the static detection

| | Resolved files | | | Context-sensitive Emulation | | | | | Unsafe loadings / Specifications | |
|---|---|---|---|---|---|---|---|---|---|---|
| | # | Size (MB) | Disasm. time | Call sites | Slices | Slice inst. (#) mean | Slice inst. (#) max | Failures | Loadtime | Runtime |
| **Acrobat Reader 9.3.2** | 18 | 38.2 | 34m 12s | 85 | 145 | 5.1 | 40 | 34 | 12 / 109 | 40 / 111 |
| **Firefox 3.0** | 13 | 12.5 | 10m 48s | 21 | 25 | 2.7 | 26 | 3 | 9 / 77 | 12 / 22 |
| **iTunes 9.0.3** | 2 | 25.1 | 11m 32s | 53 | 128 | 13.7 | 187 | 74 | 18 / 36 | 31 / 54 |
| **Opera 10.50** | 3 | 11.6 | 12m 46s | 28 | 30 | 3.0 | 29 | 2 | 8 / 28 | 11 / 28 |
| **Quicktime 7.6.5** | 17 | 40.5 | 9m 15s | 70 | 119 | 13.5 | 54 | 58 | 19 / 109 | 19 / 61 |
| **Safari 5.31** | 24 | 37.5 | 11m 03s | 72 | 137 | 5.8 | 48 | 33 | 16 / 158 | 67 / 104 |
| **Seamonkey 2.0.4** | 15 | 14.5 | 20m 44s | 34 | 40 | 1.7 | 24 | 2 | 9 / 88 | 20 / 38 |
| **Thunderbird 3.0.4** | 15 | 15.0 | 19m 38s | 34 | 40 | 1.7 | 24 | 2 | 9 / 88 | 20 / 38 |
| **Foxit Reader 3.0** | 2 | 10.2 | 5m 20s | 18 | 18 | 2.1 | 13 | 5 | 10 / 24 | 6 / 13 |

**Table 2.** Analysis of scalability

| Software | Detection time | | | | | Relative cost of slice construction | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | # of analyzed functions | | | | # of inst. of analyzed functions | | | |
| | Open (s) | Call site (s) | Slicing (s) | Emulation (s) | Total (s) | Demand-driven mean | Demand-driven max | Demand-driven total | Static total | Demand-driven mean | Demand-driven max | Demand-driven total | Static total |
| **Acrobat Reader 9.3.2** | 95.68 | 0.03 | 3.11 | 6.17 | 104.93 | 1.4 | 3 | 205 | 264,551 | 48.4 | 220 | 7,019 | 9,907,069 |
| **Firefox 3.0** | 41.69 | 0.03 | 0.19 | 0.22 | 42.13 | 1.0 | 1 | 25 | 63,550 | 34.4 | 158 | 859 | 3,071,548 |
| **iTunes 9.0.3** | 15.47 | 0.03 | 23.53 | 16.80 | 55.83 | 2.2 | 5 | 280 | 42,689 | 222.3 | 7,017 | 28,460 | 3,612,724 |
| **Opera 10.50** | 15.35 | 0.03 | 0.20 | 0.57 | 16.15 | 1.0 | 1 | 30 | 54,387 | 28.1 | 140 | 843 | 2,789,126 |
| **Quicktime 7.6.5** | 46.70 | 0.02 | 4.65 | 25.64 | 77.01 | 1.9 | 7 | 221 | 63,995 | 84.4 | 1,542 | 10,038 | 4,885,911 |
| **Safari 5.31** | 48.34 | 0.02 | 1.96 | 3.70 | 54.02 | 1.5 | 7 | 201 | 80,899 | 49.5 | 500 | 6,788 | 5,058,285 |
| **Seamonkey 2.0.4** | 37.51 | 0.02 | 0.19 | 0.52 | 38.24 | 1.0 | 1 | 40 | 79,636 | 30.9 | 125 | 1,236 | 3,840,465 |
| **Thunderbird 3.0.4** | 37.22 | 0.02 | 0.22 | 0.53 | 37.99 | 1.0 | 1 | 40 | 78,520 | 30.9 | 125 | 1,236 | 3,782,799 |
| **Foxit Reader 3.0** | 12.08 | 0.01 | 0.17 | 0.28 | 12.54 | 1.2 | 3 | 22 | 56,439 | 22.8 | 72 | 411 | 2,032,545 |

According to our analysis of context-sensitive emulation, the number of slices is generally larger than that of the call sites. This indicates that parameters for loading library calls can have multiple values, confirming the need for context-sensitive slices. The average number of instructions for the slices is quite small, which empirically validates our analysis design decisions.

We now discuss the evaluation of our tool's scalability. To this end, we measure its detection time and the efficiency of its backward slicing phase. Table 2 shows the detailed results of detection time and relative cost of slice construction. The results show that our analysis is practical and can analyze all nine large applications within minutes. To further understand its efficiency, we compared cost of our backward slicing with one of standard SDG-based slicing. Although we do expect to explore fewer instructions with a demand-driven approach, we include the comparison in Table 2 to provide concrete, quantitative data. For a standard SDG-based approach, one has to construct the complete SDG before performing slicing. We thus measured how many functions and instructions there are in each application as these numbers indicate the cost of this *a priori* construction (*cf.* the two columns labeled "Static total"). As the table shows, we achieve orders of magnitude reduction in terms of both the number of functions and the number of instructions analyzed.

**Comparison with Dynamic Detection.** To evaluate our tool's code coverage, we compare unsafe loadings detected by the static and dynamic analyses. In particular, we detected unsafe component loadings with the existing dynamic technique [21] and compared its results with our static detection. To collect the runtime traces, we executed

**Table 3.** Static detection versus dynamic detection [21]

| Software | Component loadings | | | Unsafe loadings | | | Static reachability | |
|---|---|---|---|---|---|---|---|---|
| | Dynamic | Static | ∩ | Dynamic | Static | ∩ | Reachable | Unknown |
| **Acrobat Reader 9.3.2** | 14 | 111 | 11 | 2 | 40 | 1 | 32 | 7 |
| **Firefox 3.0** | 16 | 22 | 11 | 6 | 12 | 4 | 1 | 7 |
| **iTunes 9.0.3** | 5 | 54 | 2 | 3 | 31 | 1 | 29 | 1 |
| **Opera 10.50** | 20 | 28 | 13 | 9 | 11 | 4 | 7 | 0 |
| **Quicktime 7.6.5** | 6 | 61 | 4 | 2 | 19 | 1 | 9 | 9 |
| **Safari 5.31** | 27 | 104 | 24 | 17 | 67 | 15 | 52 | 0 |
| **Seamonkey 2.0.4** | 24 | 38 | 12 | 9 | 20 | 6 | 0 | 14 |
| **Thunderbird 3.0.4** | 25 | 38 | 11 | 6 | 20 | 5 | 0 | 15 |
| **Foxit Reader 3.0** | 6 | 13 | 1 | 0 | 6 | 0 | 6 | 0 |

our test subjects one by one with relevant inputs (*e.g.*, PDF files for Acrobat Reader) and collected a single trace per application. Please note that the dynamically detected unsafe loadings are only a subset of all real unsafe loadings.

In this evaluation, we focus on application-level runtime unsafe loadings as loadtime dependent components are loaded by OS-level code. Table 3 shows the detailed results. We see that our static analysis can detect not only most of the dynamically-detected unsafe loadings but also many additional (potential) unsafe loadings, most of which we believe are real and should be fixed. Next we closely examine the results.

***Static-only Cases.*** Our static analysis detects many additional potential unsafe loadings. We carefully studied these additional unsafe loadings manually. In particular, we analyzed whether they are reachable from the entry points of the programs, *i.e.*, whether there exist paths from the entry points to the call sites of the unsafe loadings in the programs' interprocedural CFGss (ICFGs). In this analysis, we consider the main function of an application and the UI callback functions as the entry points of the application's ICFG. Table 3 shows our results on this reachability analysis. Note that those loadings marked as "Unknown" may still be reachable as it is difficult to resolve indirect jumps in binary code, so certain control flow edges may be missing from the ICFGs. All the statically reachable unsafe loadings lead to component-load hijacking if 1) the corresponding call sites are invoked and 2) the target components have not been loaded yet.

Although it is difficult to trigger the detected call sites dynamically (due to the size and complexity of the test subjects), we believe most of the call sites are *dynamically reachable* as dead-code is uncommon in production software. As a concrete example of unsafe loading, Foxit Reader 3.0 has a call site for loading `MAPI32.DLL`, which is invoked when the current PDF file is attached to an email message. This loading can be hijacked by placing a file with the same name `MAPI32.DLL` into the directory where Foxit Reader 3.0 is installed.

***Dynamic-only Cases.*** According to Table 3, our technique misses a few of the dynamically detected unsafe loadings. We manually examined all these cases, and there are two reasons for this: *system hook dependency* and *failed emulation*, which we elaborate next.

First, Microsoft Windows provides a mechanism to hook particular events (*e.g.*, mouse events). If hooking is used, a component can be loaded into the process to handle the hooked event. This component injection introduces a system hook dependency [41]. Such a loading may be unsafe, but since it is performed by the OS at runtime and is not an application error, we do not detect it.

Second, our extraction phase may miss some target component specifications due to failed emulations. If this happens, we may miss some unsafe loadings even if their corresponding call sites are found. Emulation failures can be caused by the following reasons.

*External Parameters.* A target specification may be defined by a parameter of an exported function that is not invoked. For example, suppose that a function `foo` exported by a component `A` loads a DLL specified by `foo`'s parameter. If `foo` is not invoked by `A`, the parameter's concrete value will be unknown. One may mitigate this issue by analyzing the data flow dependencies among the dependent components. However, such an analysis does not guarantee to obtain all the target specifications, because the exported functions are often not invoked by the dependent components.

*Uninitialized Memory Variables.* The slices may have instructions referencing memory variables initialized at runtime. In this case, our slice emulation may be imprecise or fail. To address this problem, it is necessary to extract the sequence of instructions from the dependent components that initialize these memory variables and emulate the instructions before slice emulation. Although it is possible to analyze memory values, such as the Value Set Analysis (VSA) [34], it is difficult to scale such analysis to large applications.

*Imprecise Inferred Function Prototypes.* Our technique infers function prototypes by analyzing parameters passed via the stack. However, function parameters may be passed via other means such as registers. For example, the `__fastcall` convention uses `ECX` and `EDX` to pass the first two parameters. Therefore, when function parameters are passed through unsupported calling conventions, the inferred function prototypes may omit parameters that determine the new slicing criteria. For example, suppose that we extract a context-sensitive sub-slice $s$ from a function `foo`, and `ECX` is used as a parameter variable of $s$. In this case, we do not continue the backward slicing phase, because the inferred prototype does not contain `ECX`. Although imprecisely inferred function prototypes may lead to emulation failure, our results show that this rarely happens in practice—we observed only 14 cases out of a total of 213.

*Unknown Semantics of System Calls.* Detailed semantics of system calls is often undocumented, and sometimes even their names are not revealed. When we encounter such system calls, we cannot analyze nor emulate them. When information of such system calls becomes available, we can easily add analysis support for them.

*Disassemble Errors.* Our implementation relies on IDA Pro to disassemble binaries, and sometimes the disassemble results are incorrect. For example, IDA Pro sometimes is not able to disassemble instructions passing parameters to call sites for delayed loading. Such errors can lead to imprecise slices and emulation failures.

## 4 Related Work

We survey additional related work besides the one on dynamic detection of unsafe loadings [21], which we have already discussed.

Our technique performs static analysis of binaries. Compared to the analysis of source code, much less work exists [1, 3, 4, 6, 7, 19, 20, 34, 39]. In this setting, Value Set Analysis (VSA) [3, 34] is perhaps the most closely related to ours. It combines numeric and pointer analyses to compute an over-approximation of numerical values of program variables. Compared to VSA, our technique focuses on the computation of

string variables. It is also demand-driven and uses context-sensitive emulation to scale to real-world large applications.

Starting with Weiser's seminal work [43], program slicing has been extensively studied [40, 46]. Our work is related to the large body of work on static slicing, in particular the SDG-based interprocedural techniques. Standard SDG-based static slicing techniques [1, 5, 13, 30, 35, 38] build the complete SDGs beforehand. In contrast, we build control- and data-flow dependence information in a demand-driven manner, starting from the given slicing criteria. Our slicing technique is also modular because we model each call site using its callee's inferred summary that abstracts away the internal dependencies of the callee. In particular, we treat a call as a non-branching instruction and approximate its dependencies with the callee's summary information. This optimization allows us to abstract away detailed data flow dependencies of a function using its corresponding call instruction. We make an effective trade-off between precision and scalability. As shown by our evaluation results, function prototype information can be efficiently computed and yield precise results for our setting.

Our slicing algorithm is demand-driven, and is thus also related to demand-driven dataflow analyses [14, 33], which have been proposed to improve analysis performance when complete dataflow facts are not needed. These approaches are similar to ours in that they also leverage caller-callee relationship to rule out infeasible dataflow paths. The main difference is that we use a simple prototype analysis to construct concise function summaries instead of directly traversing the functions' intraprocedural dependence graphs, *i.e.*, their PDGs. Another difference is that we generate context-sensitive executable slices for emulation to avoid the difficulties in reasoning about system calls.

As we discussed earlier, instead of emulation, symbolic analysis [18, 37] could be used to compute concrete values of the program variables. However, symbolic techniques generally suffer from poor scalability, and more importantly, it is not practical to symbolically reason about system calls, which are often very complex. The missing implementation for undocumented system calls is the challenge for emulation, while for symbolic analysis, complex system call implementation is an additional challenge. We introduce the combination of slicing and emulation to address this additional challenge. Our novel use of context-sensitive emulation provides a practical solution for computing the values of program variables.

## 5 Conclusion and Future Work

We have presented a practical static binary analysis to detect unsafe loadings. The core of our analysis is a technique to precisely and scalably extract which components are loaded at a particular loading call site. We have introduced context-sensitive emulation, which combines incremental and modular slice construction with the emulation of context-sensitive slices. Our evaluation on nine popular Windows application demonstrates the effectiveness of our technique. Because of its good scalability, precision, and coverage, our technique serves as an effective complement to dynamic detection [21]. For future work, we would like to consider two interesting directions. First, because unsafe loading is a general concern and also relevant for other operating systems, we plan to extend our technique and analyze unsafe component loadings on Unix-like systems. Second, we plan to investigate how our technique can be improved to reduce emulation failures.

**Acknowledgments.** We thank the anonymous reviewers for their feedback on an earlier version of this paper. This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, NSF TC Grant No. 0917392, and the US Air Force under grant FA9550-07-1-0532. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

## References

1. Kiss, Á., Jász, J., Lehotai, G., Gyimóthy, T.: Interprocedural static slicing of binary executables. In: Proc. SCAM Workshop (2003)
2. An update on the DLL-preloading remote attack vector, `http://blogs.technet.com/b/srd/archive/2010/08/31/an-update-on-the-dll-preloading-remote-attack-vector.aspx`
3. Balakrishnan, G., Reps, T.: Analyzing Memory Accesses in x86 Executables. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 5–23. Springer, Heidelberg (2004)
4. Balakrishnan, G., Reps, T.: Analyzing Stripped Device-Driver Executables. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 124–140. Springer, Heidelberg (2008)
5. Binkley, D.: Precise executable interprocedural slices. ACM Lett. Program. Lang. Syst. 2(1-4), 31–45 (1993)
6. Cifuentes, C., Fraboulet, A.: Intraprocedural static slicing of binary executables. In: Proc. ICSM (1997)
7. Comparetti, P.M., Salvaneschi, G., Kirda, E., Kolbitsch, C., Kruegel, C., Zanero, S.: Identifying dormant functionality in malware programs. In: Proc. SSP (2010)
8. dlopen man page, `http://linux.die.net/man/3/dlopen`
9. Dynamic-Link Library Search Order, `http://msdn.microsoft.com/en-us/library/ms682586(VS.85.)aspx`
10. Dynamic-Link Library Security, `http://msdn.microsoft.com/en-us/library/ff919712(VS.85.)aspx`
11. Exploiting DLL Hijacking Flaws, `http://blog.metasploit.com/2010/08/exploiting-dll-hijacking-flaws.html`.
12. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Prog. Lang. Syst. 9(3), 319–349 (1987)
13. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Prog. Lang. Syst. 12(1), 26–60 (1990)
14. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. In: Proc. FSE (1995)
15. IDA Pro Disassmelber, `http://www.hex-rays.com/idapro/`
16. IDAPython, `http://code.google.com/p/idapython/`
17. Insecure Library Loading Could Allow Remote Code Execution, `http://www.microsoft.com/technet/security/advisory/2269637.mspx`
18. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
19. Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: Proc. USENIX Security (2004)
20. Kruegel, C., Robertson, W., Vigna, G.: Detecting Kernel-Level Rootkits Through Binary Analysis. In: Proc. ACSAC (2004)
21. Kwon, T., Su, Z.: Automatic detection of unsafe component loadings. In: Proc. ISSTA (2010)
22. Kwon, T., Su, Z.: Static detection of unsafe component loadings. UC Davis techical report CSE-2010-17 (2010)

23. Microsoft Cooking Up Baker's Dozen of Fixes for Patch Tuesday, http://www.esecurityplanet.com/patches/article.php/3902856/Microsoft-Cooking-Up-Bakers-Dozen-of-Fixes-for-Patch-Tuesday.htm
24. Microsoft Portable Executable and Common Object File Format Specification, http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx
25. Microsoft releases tool to block DLL load hijacking attacks, http://www.computerworld.com/s/article/print/9181518/Microsoft_releases_tool_to_block_DLL_load_hijacking_attacks
26. Microsoft releases tool to block DLL load hijacking attacks, http://www.computerworld.com/s/article/9181518/Microsoft_releases_tool_to_block_DLL_load_hijacking_attacks
27. Microsoft Was Warned of DLL Vulnerability a Year Ago, http://www.esecurityplanet.com/features/article.php/3900186/Microsoft-Was-Warned-of-DLL-Vulnerability-a-Year-Ago.htm
28. MS09-014: Addressing the Safari Carpet Bomb vulnerability, http://blogs.technet.com/srd/archive/2009/04/14/ms09-014-addressing-the-safari-carpet-bomb-vulnerability.aspx
29. NetworkX, http://networkx.lanl.gov/
30. Orso, A., Sinha, S., Harrold, M.J.: Incremental slicing based on data-dependence types. In: Proc. ICSM (2001)
31. pefile, http://code.google.com/p/pefile/
32. PyEmu, http://code.google.com/p/pyemu/
33. Reps, T.: Solving Demand Versions of Interprocedural Analysis Problems. In: Adsul, B. (ed.) CC 1994. LNCS, vol. 786, pp. 389–403. Springer, Heidelberg (1994)
34. Reps, T., Balakrishnan, G.: Improved Memory-Access Analysis for x86 Executables. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 16–35. Springer, Heidelberg (2008)
35. Reps, T., Horwitz, S., Sagiv, M., Rosay, G.: Speeding up slicing. In: Proc. FSE (1994)
36. Researcher told Microsoft of Windows apps zero-day bugs 6 months ago, http://www.computerworld.com/s/article/print/9181358/Researcher_told_Microsoft_of_Windows_apps_zero_day_bugs_6_months_ago
37. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proc. SSP (2010)
38. Sinha, S., Harrold, M.J., Rothermel, G.: System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In: Proc. ICSE (1999)
39. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A New Approach to Computer Security via Binary Analysis. In: Sekar, R., Pujari, A.K. (eds.) ICISS 2008. LNCS, vol. 5352, pp. 1–25. Springer, Heidelberg (2008)
40. Tip, F.: A survey of program slicing techniques. Technical report, CWI, Amsterdam, The Netherlands (1994)
41. Types of Dependencies, http://dependencywalker.com/help/html/dependency_types.htm
42. Vulnerabilities in Microsoft Office Could Allow Remote Code Execution, http://www.microsoft.com/technet/security/bulletin/ms10-087.mspx
43. Weiser, M.: Program slicing. In: Proc. ICSE (1981)
44. Windows DLL Exploits Boom; Hackers Post Attacks for 40-plus Apps, http://www.computerworld.com/s/article/9181918/Windows_DLL_exploits_boom_hackers_post_attacks_for_40_plus_apps
45. X86 Calling Conventions, http://en.wikipedia.org/wiki/X86_calling_conventions
46. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. SIGSOFT Softw. Eng. Notes 30(2), 1–36 (2005)