Static Detection of Unsafe Component Loadings on Windows and Linux: A Survey

Sneha D. Patel
PG Student
Department of Computer Engineering, NMU
SES's R. C. Patel Institute of Technology, Shirpur

Tareek M. Pattewar
Assist. Proffesor
Department of Information Technology, NMU
SES's R. C. Patel Institute of Technology, Shirpur

ABSTRACT
Dynamic loading is an essential mechanism for computer software development. It enables an program, the versatility to use its exported functionalities and energetically link a part. Dynamic loading is really a system by which a pc program are able to at run-time, fill a collection into memory, recall the handles of parameters and functions included in the library, run those functions or get those variables, and sell the library from recollection. This function presents a signal coverage approach called motionless binary analysis to assess and discover mistakes and weaknesses about the element. Thus the dangerous and exposed parts may be recognized previous to loading energetically into applications. 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. In particular, for a given call site, 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 call site. For evaluation, we implemented our technique to detect vulnerable and unsafe component loadings in popular software on Microsoft Windows and Linux.

General Terms
Dynamic loading, Unsafe Component Loading, Vulnerability

Keywords
Software Engineering, Component Testing, Regression Testing, Code coverage

1. INTRODUCTION
Dynamic element load is utilized in software development to develop and adaptable software. Java run-time environment (JRE) generally supplies applicable method calls to fill dynamic parts. The inherent JRE solves and lots the given element, once a launching system phone is invoked. Component resolution is dependent upon the way the part is specified moreover through the plan part's full path or its file-name. Given a complete route, the JAVA Runtime Environment just uses it for quality. Which series of sites to search is managed at run-time by the special directory explore order at that instance of program call invocation. The versatility of the typical fashion of element loading does include a cost an inherent security apprehension is introduced by it.

For runtime security and protection, and request should just fill its planned parts. Nevertheless, as a constituent is resolved by the JRE only during its name, programming errors may cause the launching of an accidental component with exactly the same name. An approach was suggested to find dangerous part loadings. It then performs a evaluation to discover two kinds of dangerous loadings: resolution and resolution failure hijacking. When the target part isn't discovered, though a resolution hijacking occurs when other sites are looked before the listing where the part lives a quality failure occurs. We illustrate this dilemma using delayed loading, an optimization to delay the loading of rarely used parts until their very first use. Since it is hard to activate all deferred loadings at runtime delayed loading is tough for dynamic detection. Within this, document the very first static analysis to find dangerous loadings from program binaries.

Two items of essential advice are needed

1) All parts which may be packed at every loading call website, and
2) The security of each potential loading. From these findings,

We style a two period analysis: checking and extraction. The removal stage is demand driven, working backwards from every loading call website to calculate the group of potential loadings; the stage establishes the security of the loading by analyzing the applicable directory explore order in the identify site.

1.1 Context-Sensitive Emulation
We introduce context sensitive emulation, a new blend of emulation and segmenting, to comprehend the diffident computation of limitation values throughout the removal period. For a specified call site, we remove its context susceptible executable blocks in respect to its guidelines, one for every execution context. We subsequently copy the blocks to calculate the restriction values.

1.1.1 Incremental and Modular Segmenting
One specialized hurdle is the way to calculate diffident blocks scalable. Normal segmenting approaches are centered on processing a program's entire system dependency graph (SDG) a priori and are consequently restricted in scalability. Because we just have to think about loading call websites as well as the execution pathways to calculate the limitation values to the describes are generally fairly short, only a little part of the entire SDG is applicable for our evaluation. This inspires the utilization of an step-by-step and modular sectioning algorithm incremental because we construct the blocks lazily when needed; modular because when we see a perform call foo(x,y), author use an conditional outline about what addiction foo's parameters and revisit value have in examining the caller.

1.1.2 Emulation of Context-sensitive Slice
Once author calculated the piece s regarding a specified loading call website, we must calculate values for the important guidelines. One organic remedy would be to execute conventional representative analysis on the piece to calculate the ideals. The chief problem for this strategy is the issue in reasoning symbolically about method calls since the applicable

parameters frequently rely on complicated, low level system calls. [2]

2. RELATED WORK

Static Detection of Unsafe Component Loadings on Windows and Linux technique performs static analysis of binaries. Compared to the analysis of source code, much less work exists. In this setting,

Value Set Analysis (VSA) 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.[4] [5]

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. As we discussed earlier, instead of emulation, symbolic analysis 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. Our novel use of context sensitive emulation provides a practical solution for computing the values of program variables. [6] Starting with Weisers seminal work [10], program slicing has been extensively studied. Our work is related to the large body of work on static slicing, in particular the SDG-based techniques. Standard SDG-based static slicing techniques 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 callees 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 callees 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 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 intra procedural dependence graphs, i.e., their PDGs. Another difference is that we generate context-sensitive executable program slices for emulation to avoid the difficulty in reasoning about system calls.

3. METHODOLOGY

The proposed system is Static Detection of Unsafe Component Loadings. This 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 check their safety.

3.1 Extraction Phase

A component can load other components at load time or runtime. This loading introduces load time and runtime dependencies among components. Based on these dependencies, it determines components that can be loaded during program execution. Specifically, it recursively resolve the components from the program file based on their load time and runtime dependencies. To resolve the dependent components, the corresponding target specifications, i.e., full path or file name, are needed. For load time dependencies, compilers specify the dependent components in the executable format. For example, the names of the load time dependent components are stored in IMAGE_IMPORT_DIRECTORY with the PE format. To obtain the specifications of the runtime dependent components, it computes values of parameters to component-loading system calls.. As an example of recursive resolution, it search the components that can be loaded by Program in Figure 1. Suppose that components E and F, which have no load time and runtime dependent component, implement the rand and Load Library functions, respectively. The key step of the extraction phase is to obtain the target specification for component loading in a binary. The specification of a load time 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.

3.1.1 Searching Program Variable for Specification

In binary code, invoking the component-loading system calls often follows the stdcall calling convention1. When parameters are passed to the call site, they are pushed from right to left. For example, Figure 2(a) represents the binary code corresponding to LoadLibraryExA(0x7D61AC5C, EAX, EAX). Based on the parameter passing mechanism, it locates the program variable, e.g., a register or a memory chunk, which stores the target specification. In particular, it 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.

3.1.2 Locating Component-loading Call Sites

In this phase, it aim at finding the call site for component loading in a binary. In this observation the 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 2 shows the two types of component-loading call sites in a binary, which are memory indirect and register indirect. While the memory indirect type stores address in a register, e.g., line 4 in Figure 2(a) and line 3 in Figure 2(b).

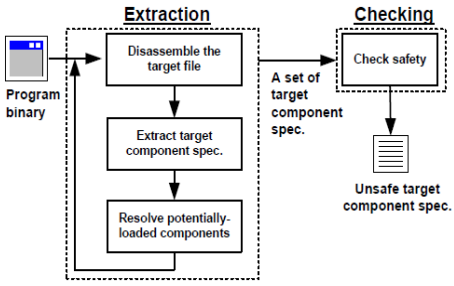


Figure 1 Detection framework

```
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.

Figure 2 Two types of component-loading call sites.

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

1. **Taint source** An instruction that references a memory chunk that stores the address of the component-loading system call.
2. **Taint sink** A branch instruction, e.g., call, whose target address is tainted.

It considers the taint sink instructions as the call sites. It presents examples on how to detect call sites. In Figure 2(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 2(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.

3.1.2 Extracting Parameter Variables

Once a call site is located, it extracts the program variables for the target specification from the predefined number of the instructions to pass the parameters to the call site. In particular, it detects 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, it can precisely extract all the variables to define this target specification. For example, the call site in Figure 2(a) invokes LoadLibraryExA, and it has three parameters, i.e., 0x7D61AC5C, EAX, and EAX, via the instructions on lines 13.

3.2 Context-sensitive Emulation

In this phase, it compute the concrete values of the parameter variables extracted. The computation may seem trivial at first. For example, the memory chunk at 0x7D61AC5C in Figure 2(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. To address this problem, it introduce context-sensitive emulation, which novely combines backward slicing and emulation. Based on this combination, it can scalably and precisely compute the values of the variables of interest.

3.2.1 Slicing

Program slicing is a method for automatically decomposing programs by analyzing their data flow and control flow. Starting from a subset of a program’s behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, is called as ”slice,”.

3.2.2 Backward Slicing

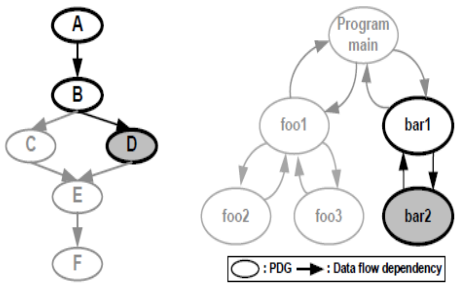
This phase performs the interprocedural backward slicing w.r.t. the parameter variable, extracting the instructions to compute the variable. This problem has been extensively studied, and many slicing algorithms have been proposed. These algorithms commonly solve the graph reachability problem over a System Dependence Graph (SDG), a set of Program Dependence Graphs (PDGs) 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 this problem setting. Thus, exhaustive data flow analysis is not be necessary to extract backward slices w.r.t. the given slicing criteria. Figure 3 shows the examples of the unnecessary data flow analysis during intraprocedural and interprocedural backward slicing. Figure 3(a) shows an example of the CFG for constructing the PDG. Suppose that it performs 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. Suppose that Figure 3(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, is it not necessary to perform data flow analysis on the instructions of the grayed PDGs. [8]

3.2.3 Intraprocedural backward Slicing

For each intraprocedural backward slicing, it analyzes only the data flow dependencies among the instructions that are control dependent on the given slicing criteria. Suppose that it perform intraprocedural backward slicing w.r.t. the instruction D in the CFG shown in Figure 3(a). If this 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 sub graph w.r.t. the instruction D, it can avoid some unnecessary data flow analysis when performing slicing.

3.2.4 Interprocedural backward Slicing

As aforementioned, an exhaustive SDG construction often leads to significant amount of the unnecessary data flow analysis for interprocedural backward slicing. To address this problem, it constructs 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.



(a) Intra-backward. (b) Inter-backward.

Figure 3 Unnecessary data flow analysis

First, it is necessary to determine the new slicing criteria if the interprocedural backward slice consists of multiple intraprocedural backward slices. For example, it construct the interprocedural backward slice in Figure 3(b) by combining the two intra-backward slices extracted from functions bar1 and bar2. In this case, it needs 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. The 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. To locate this parameter variable, it use caller-callee relationship and the callees function prototype. In particular, it detects the call site for function f and analyze fs 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 to 7 and lines 14 to 16 invoke_delayLoadHelper2, it 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, it constructs 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 consists of the instructions of three intraprocedural backward slices w.r.t. the slicing criteria, i.e., target_dllname, pDelayDesc1, and pDelayDesc2. [7]

3.2.5 Function Prototype Analysis

The backward slicing phase relies on function prototypes, but such information is often unavailable in binary code. This solution to the problem is as follows. For a given function f, its parameters are stored in fixed locations during fs execution. Thus, it infers its prototype by analyzing how the instructions of the function access the memory chunks for the parameters, i.e., read or write. Figure 4 shows an example of our proposed prototype analysis for the foo function. Suppose that Figures 4(a) and 4(b) show part of foo and the stack layout at the beginning of the functions execution, respectively. To improve the precision of our prototype inference, it uses the following effective heuristic. If the effective address of the memory chunk, obtained by the lea instruction, is passed to the function, it considers it as the in out parameter. Although this heuristics may increase the size of the computed slice, it is sufficient to compute possible values of the slicing criteria via emulation. [9] [1] [3]

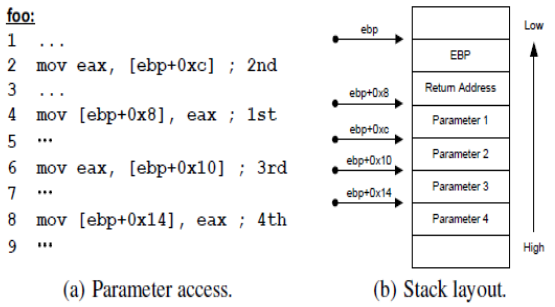


Figure 4 Function prototype analysis

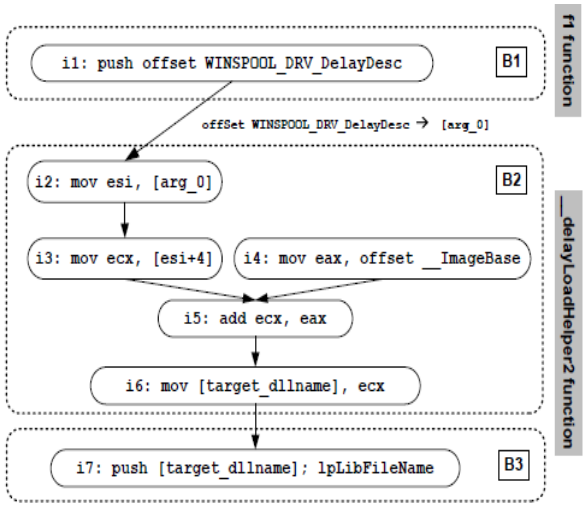


Figure 5 Data-flow dependencies among basic blocks

3.3 Emulation Phase

In this phase, it computes the possible values of the target component specification by emulating its corresponding context-sensitive slices. There are two challenges for slice emulation. The first challenge is how to schedule the instructions because it do not know their runtime execution sequence. In this case, before emulating fs basic blocks, it 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. The instructions use arg_0 to reference the first parameter. Based on the techniques mentioned, it emulates the context sensitive slices to compute the possible values of the target component specification. For example, it can compute the value, "WINSPOOL. DRV", of the target dllname by emulating the backward slice.

3.4 Checking Phase

In this phase, it evaluates the safety of the target component specifications obtained from the extraction phase. To this end, for each specification, it check whether or not the safety conditions in are satisfied. Specifically, it consider that a specification can lead to unsafe loading if the OS cannot resolve the target component in the directory that is first searched. Note that the first directory searched by the OS for the resolution is known.

4. CONCLUSION

In this, it is presented as practical static binary analysis to detect unsafe loadings. The core of this analysis is a technique to precisely and scalable extract which components are loaded at a particular loading call site. They have introduced context-sensitive emulation, which combines incremental and modular slice construction with the emulation of context-sensitive slices. This 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. For future work, we would like to consider interesting directions. First, because unsafe loading is a general concern and also relevant for other operating systems, it plan to extend our technique and analyze unsafe component loadings on Unix-like systems.

5. ACKNOWLEDGEMENT

We are thankful to all the personalities who helped us throughout this survey.

6. REFERENCES

[1] Lal A. Burton E. Driscoll M. Elder T. Andersen A. V. Thakur, J. Lim and T. W. Reps. Directed proof generation for machine code. 2010.

[2] G. Lehotai Akos Kiss, J. Jasz and T. Gyimothy**.** Interprocedural static slicing of binary executables. SCAM, 1:68–79,March 2003.

[3] X. Zhang Z. Wu B. Xu, J. Qian and L. Chen. A brief survey of program slicing. 2005.

[4] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. 2004.

[5] D. Binkley. Precise executable interprocedural slices. ACM Lett.Program. Lang, 2003.

[6] K. J. Ottenstein J. Ferrante and J. D.Warren. The program dependence graph and its use in optimization. ACM Trans, 2:23–45, June 1987

[7] T. Reps and G. Balakrishnan. Improved memory-access analysis for x86 executables. 2:376–390, 2008

[8] T. Reps S. Horwitz and M. Sagiv. Demand interprocedural dataflow analysis.

[9] J. Lim T. Reps, G. Balakrishnan and T. Teitelbaum. A nextgeneration platform for analyzing executables. 2005.

[10] M. Weiser. Program slicing.

[11] A. V. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. W. Reps. Directed proof generation for machine code. In Proc. CAV, 2010.

[12] T. Reps and G. Balakrishnan. Improved memory-access analysis for x86 executables. In Proc. CC, 2008.

[13] J. Lim, A. Lal, and T. Reps. Symbolic analysis via semantic reinterpretation. In Proc. SPIN, 2009.