Volume 3, Issue 7, July 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Appraising Safety of Dynamic Software Updates by Version Consistency: A Runtime Analysis Based Regression Testing Recommendations

V.Haritha*
Scholar
Dept. of Computer Science and Engineering
Madanapalle Institute of Technology and Science
P.B.No.14, Angallu, Madanapalle-517325
Chittoor (district), Andrapradesh
India

E. Madhusudhana Reddy
Professor
Dept. of Computer Science and Engineering
Madanapalle Institute of Technology and Science
P.B.No.14, Angallu, Madanapalle-517325
Chittoor (district,), Andrapradesh
India

Abstract- Dynamic loading is an essential mechanism for computer software development. It enables a 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 static detection 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.

Keywords: software evaluation, Dynamic software loaded, component bug prediction, JRE, code coverage

I. Introduction

Dynamic element load is extensively utilized in software development to develop modular 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 planned part's full path or its file-name. Provided 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, an 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. Recent work [12] has proven that dangerous loadings are common and may cause remote code execution attacks. 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, we current the very first static analysis to find dangerous loadings from program binaries. Two items of essential advice are needed

- All parts which may be packed at every loading call website, and
- 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.

A.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.

B. Incremental and Modular Segmenting

One specialized hurdle is the way to calculate diffident blocks scalable. Normal segmenting approaches [1, 5, 9, 16, 20, 21] 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 (cf. Section III)--incremental because we construct the blocks lazily when needed; modular because when we see a perform call foo(x, y), we use an conditional outline about what addiction foo's parameters and revisit value have in examining the caller. In the finish, we join the function point blocks in the conventional manner by connecting real and formal parameters.

C. Emulation of Context-Sensitive Slice

Once we've 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. To conquer this issue, we use emulation. In specific, we create, from the backward piece s, a group of context sensitive executable sub blocks, which we subsequently emulate to calculate the parameter values (cf. Area III). s's sub-blocks $s_1,...,s_n$. Instructions in every sub-slice s_i are next follow topologically, respecting their information and control-flow dependencies.

For assessment, we implemented our technique in a model program for Windows software. We assessed our tool's effectiveness beside the previous dynamic tool [12] in relation to precision, scalability, and coverage. Results on nine popular applications reveal that our device is scalable and exact (cf. Part IV). Like, it took less than two moments to examine each of the nine test subjects, including substantial programs such as Acrobat Reader, QuickTime, and Safari. The results also show that our projected context sensitive emulation attains orders of magnitude decrease in the size of the code needed to be examined and crucially supply to the scalability of our technique. In terms of coverage, our tool detected many more possible dangerous loadings and nicely matches the dynamic technique.

D. Main Contributions

We have urbanized the first static dual analysis to detect unsafe constituent loadings. Because of its scalability and superior code coverage, our procedure effectively complements the accessible dynamic technique. We have projected context-sensitive emulation, an efficient approach that combines segmenting and emulation for the accurate and scalable analysis of runtime standards of program variables. We have realized our method and evaluated its effectiveness by detecting perilous loadings in nine admired Windows applications.

The rest of this thesis is prepared as follows. Section II illustrates our method with a running example. Section III presents a comprehensive description of our static recognition algorithm. We describe our implementation and assessment in Section IV. Finally, Section V surveys extra related work, and Section VI concludes with a conversation of future work.

II. Overview

This segment illustrates our method. Our method works on binaries, but for presentational reason, we show the example in C-like pseudo code.

A.Extraction Phase

We foremost identify call sites for constituent loading. In the example, line 23 corresponds to a call site as of the Load Library system call. The system call's only limitation target_ API determines which constituent should be loaded. We use context-sensitive emulation to calculate its possible values.

1) Incremental and Modular Segmenting: Program segmenting typically considers manage stream dependencies and data to extract a slice. In our setting, since the primary goal is to compute possible values of target_ API, we create the piece and concentrate on data dependencies. To compute the possible ideals of goal_ API, we need to take out the code that figures the foremost parameter of the function. To the end, we maintain the backward segmenting in respect to your new segmenting standard, which is established based on caller-callee relationship and also the callee's function image. In our example, there survive two call sites. Consequently, we continue with two instances of Intra technical backward segmenting in respect to two original segmenting criteria. We create two context sensitive inter procedural blocks by instantiating twice the slice for delay and linking each instance with its various caller's slice. We also maintain the maps between each of the new segmenting criteria and the callee's equivalent parameters for the brusquely emulation period. We terminate the segmenting computation, because neither of takes any input signal.

2) Emulation of flow related blocks: The two blocks are followed by us, to calculate values for target _ API. We must timetable the guidelines in the blocks previous to they could be copied. We do so regarding the data and manage flow dependencies between the instructions. Specifically, we first routine the fundamental blocks in topological regulate with respect to the information flow dependencies between them. We subsequently establish the ordering of the guidelines in each programmed basic block in relation to their organize flow dependencies confine in the original code.

B. Checking Phase

When the JRE loads the components, it iterates during a series of directories, determined at run-time, to locate the files. In this situation, these consignments are dangerous, if the JRE checks several sites to solve these parts. This is as these loadings could be hijacked by putting an arbitrary document. We examine whether or not the given files exist in the primary directory searched. Because MS Windows searches foremost in the directory anywhere the program is installed [7], the loadings for these two parts are unsafe if they cannot survive in the program directory.

III. Static Detection Algorithm

In this segment, we present background in sequence on unsafe constituent loadings and details of our analysis.

A.Background

Dynamic constituent loading is frequently supported by java runtime surroundings through meticulous system calls that acquire as input a full path or file name for the projected component. The situation of determining the target constituent by JRE as follows:

- The object constituent can be scrupulous by its full path or its class.
- When full conduit is used, the JRE openly determines the target using the complete full path.
- Otherwise, if file name is worn and recognized by the JRE, the full path of the scrupulous class is predefined.
- If the individual file name is unidentified to the JRE, it iterates during the predefined class paths to locate the first file with the scrupulous file name.

To sanctify the constituent resolution process, it is necessary to model the class path state, because even the similar component- loading code may effects in dissimilar resolutions under dissimilar class path states.

Component Resolution: A constituent resolution function γ obtains a constituent requirement $f \in \sum^{*}$, a directory search order $d = (d_{1,\ldots d_n}) \in \Sigma^* \times \ldots \Sigma^*$ $and\ a\ class\ path$ state σ and precedes a determined full path $\pi \in \Sigma^*$, where Σ represents the alphabet used to identify files and indexes.

If f is a full path, $\gamma(f,d,\sigma) = \{$ f if $f \in \sigma$; $\in$ or else where $\in$ is the empty string

If *f* is a file name,

$\gamma(f,d,\sigma) = \{\ \pi$ If f is recognized to the JRE as; $\in$ or else, where *"+ "* indicates string concatenation

We next celebrate component loading, for which we necessitate to consider the currently loaded mechanism. The enthusiasm is that the JRE does not load the same constituent several times. In our formalization, we let the position of encumbered components *L* be the set of complete paths of all the currently loaded components.

*Component Loading: p*articular the loaded components L, a constituent loading function takes a constituent condition $f \in \Sigma^*, a$ directory explore order $d = (d_{1,\ldots d_n})$ $\in \Sigma^* \times \ldots \Sigma^*$ a file system state a, and the position of loaded components *L*, and proceeds a declaration success or failure:

$\lambda(f,d,\sigma,L) = \{$ Success if $\gamma(f,d,\sigma) \notin \{\in\} \cup L$ Failure

Otherwise

The dignified constituent loading mechanism is frequently used on major java runtime surroundings. Although a full path completely determines the object constituent, for a file name, the full path of the loading constituent frequently depends on the here file system state. This apparatus can lead to two types of insecure loadings: *declaration failure* and *resolution skyjacking.*

Resolution Failure: A declaration failure happens if $\gamma(f,d,\sigma) = \in$. In this container, with a complete path specification f, a capricious file with the same absolute path f can hijack the constituent loading. If f is file name, one be able to hijack this loading by insertion a file with the scrupulous name f in any directory d_i particular by the explore order $d\ = (d_{1,\ldots d_n})$

Resolution Hijacking: A resolution hijacking ensue if the consequent conditions hold

- f is the file name of the target constituent and indefinite to the JRE
- $\gamma(f,d,\sigma) = d_k + \backslash + f < k > 1$ and
- $\lambda(f,d,\sigma,L)$ = success. In this case, one can hijack the loading by placing a file with the scrupulous name f in any directory d_i where i < k.

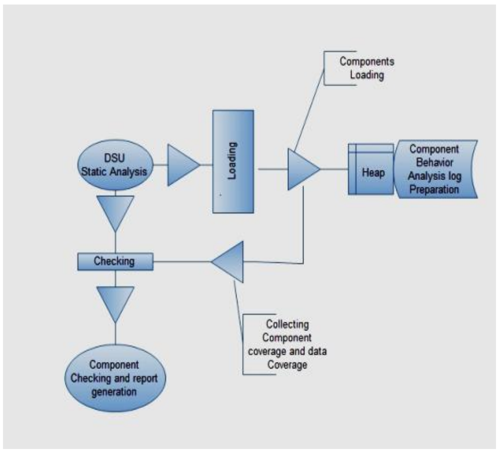

Components Loading
DSU Static Analysis
Loading
Heap
Component Behavior Analysis log Preparation
Checking
Collecting Component coverage and data Coverage
Component Checking and report generation

Fig 4 Architecture of the proposed framework

To pass up unsafe loadings, it is essential for developers to specify the objective component in a safe manner. We describe safe target constituent specifications as follows.

IV. Empirical Evaluation

In this segment, we evaluate our static method in terms of precision, scalability, and system coverage. We show that our method scales to large real-world submission and is precise. It also has good reporting, substantially better than the accessible dynamic approach [12].

A. Implementation

The semi mechanical dynamic software update assessment projected is evaluated under java run time environment. In this consider the model has been applied to test the on open source software entitle GDOWNLOADER.

B. Evaluation Setup and Results

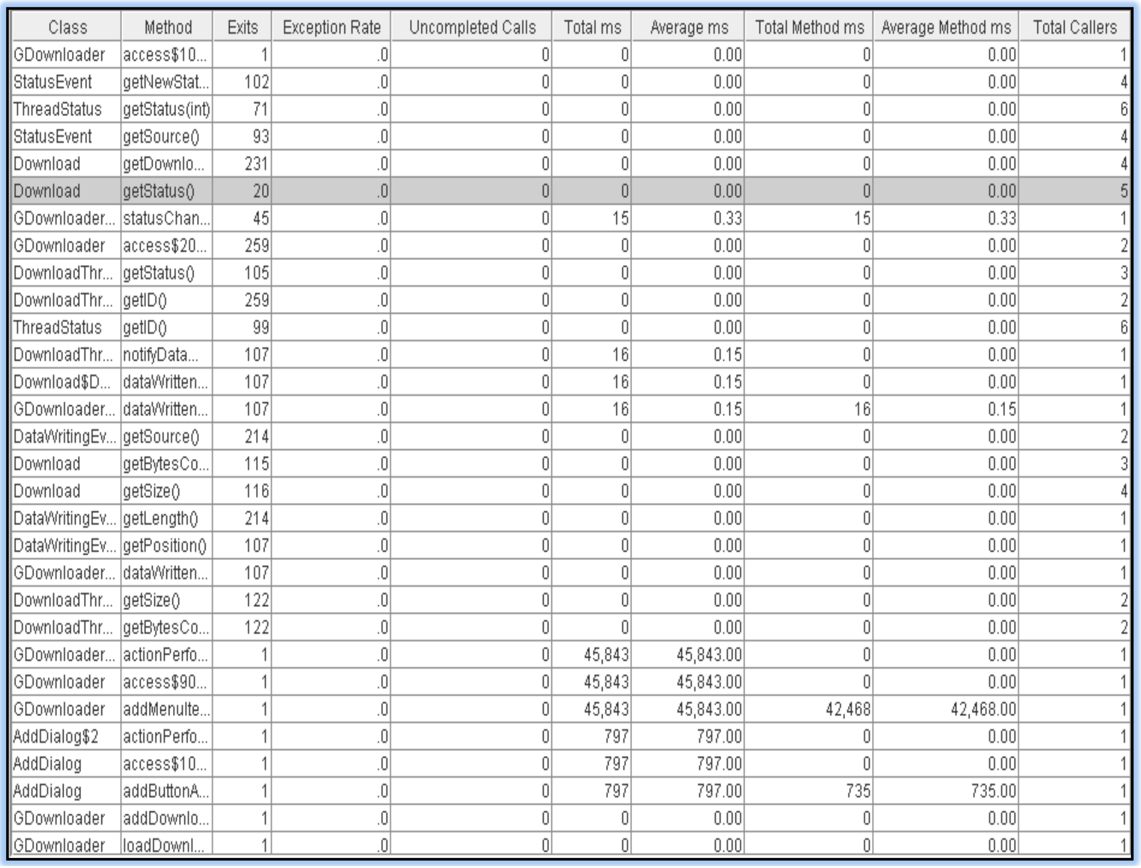
We endeavor at detecting unsafe constituent loadings in applications. Because the uncovering of insecure loadings from the API is executed by the java runtime environment, we only determine the application mechanism in the extraction phase.

1) Precision and Scalability: Table 1 shows our investigation results on eight admired Windows applications. Because they're important applications in use these applications were preferred by us as our assessment subjects. The outcomes demonstrate that our technique may effectively discover, from program binaries, dangerous constituent loadings potentially loaded at runtime. One appealing finding to notice is that the effects of the extraction phase are indistinguishable. This really is most likely because both apps are a part of the Mozilla assignment and use the exact same set of program components .As we present later our analysis time is conquered by this time. These are large software, and additionally we merely need to disassemble the code once for most of the following analysis.

According to our evaluation of context sensitive emulation, the number of blocks is typically larger than that of the contact sites. This indicates that parameters for consignment library calls can have several values, confirming the necessity for call flow related blocks. The typical number of instructions for the blocks is fairly small, which empirically authorized our evaluation design selections. We now converse the assessment of our tool's scalability. To the end, we measure its evaluation time and the effectiveness of its back-ward segmenting stage. Table1shows the comprehensive results, the results show that our analysis is sensible and can assess within minutes.

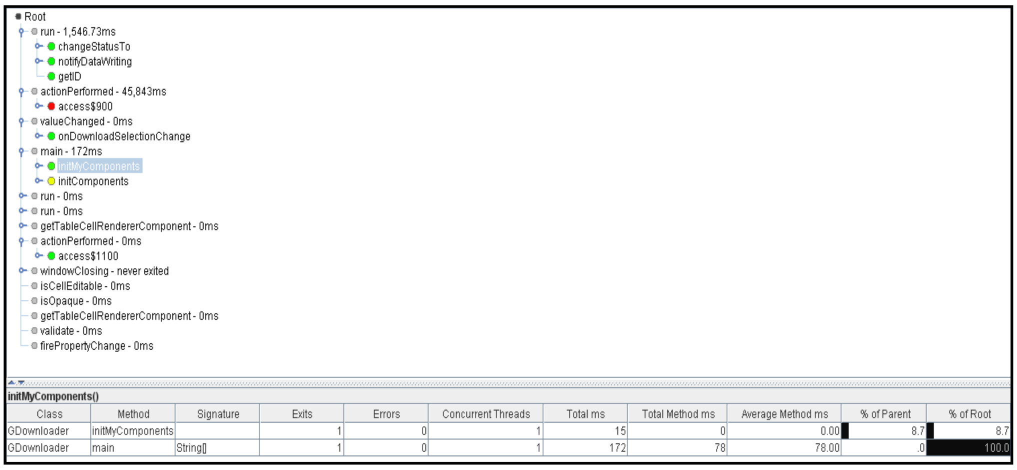
We evaluate our semi-automated DSU evaluation approach with totally mechanized and manual methods, to help understand its efficiency. We therefore measured how many guidelines and functions there are in each application because these numbers point out the price of this a priori construction. As the table1, table2 and table3 reveals, we accomplish orders of magnitude reduction in terms of both number of functions along with the number of instructions analyzed.

TABLE1
Component wise report generated by the proposed architecture



Class	Method	Exits	Exception Rate	Uncompleted Calls	Total ms	Average ms	Total Method ms	Average Method ms	Total Callers
GDownloader	access$10...	1	.0	0	0	0.00	0	0.00	1
StatusEvent	getNewStat...	102	.0	0	0	0.00	0	0.00	4
ThreadStatus	getStatus(int)	71	.0	0	0	0.00	0	0.00	6
StatusEvent	getSource()	93	.0	0	0	0.00	0	0.00	4
Download	getDownlo...	231	.0	0	0	0.00	0	0.00	4
Download	getStatus()	20	.0	0	0	0.00	0	0.00	5
GDownloader...	statusChan...	45	.0	0	15	0.33	15	0.33	1
GDownloader	access$20...	259	.0	0	0	0.00	0	0.00	2
DownloadThr...	getStatus()	105	.0	0	0	0.00	0	0.00	3
DownloadThr...	getID()	259	.0	0	0	0.00	0	0.00	2
ThreadStatus	getID()	99	.0	0	0	0.00	0	0.00	6
DownloadThr...	notifyData...	107	.0	0	16	0.15	0	0.00	1
Download$D...	dataWritten...	107	.0	0	16	0.15	0	0.00	1
GDownloader...	dataWritten...	107	.0	0	16	0.15	16	0.15	1
DataWritingEv...	getSource()	214	.0	0	0	0.00	0	0.00	2
Download	getBytesCo...	115	.0	0	0	0.00	0	0.00	3
Download	getSize()	116	.0	0	0	0.00	0	0.00	4
DataWritingEv...	getLength()	214	.0	0	0	0.00	0	0.00	1
DataWritingEv...	getPosition()	107	.0	0	0	0.00	0	0.00	1
GDownloader...	dataWritten...	107	.0	0	0	0.00	0	0.00	1
DownloadThr...	getSize()	122	.0	0	0	0.00	0	0.00	2
DownloadThr...	getBytesCo...	122	.0	0	0	0.00	0	0.00	2
GDownloader...	actionPerfo...	1	.0	0	45,843	45,843.00	0	0.00	1
GDownloader	access$90...	1	.0	0	45,843	45,843.00	0	0.00	1
GDownloader	addMenuIte...	1	.0	0	45,843	45,843.00	42,468	42,468.00	1
AddDialog$2	actionPerfo...	1	.0	0	797	797.00	0	0.00	1
AddDialog	access$10...	1	.0	0	797	797.00	0	0.00	1
AddDialog	addButtonA...	1	.0	0	797	797.00	735	735.00	1
GDownloader	addDownlo...	1	.0	0	0	0.00	0	0.00	1
GDownloader	loadDownl...	1	.0	0	0	0.00	0	0.00	1

TABLE 2
Sample call tree analysis report generated

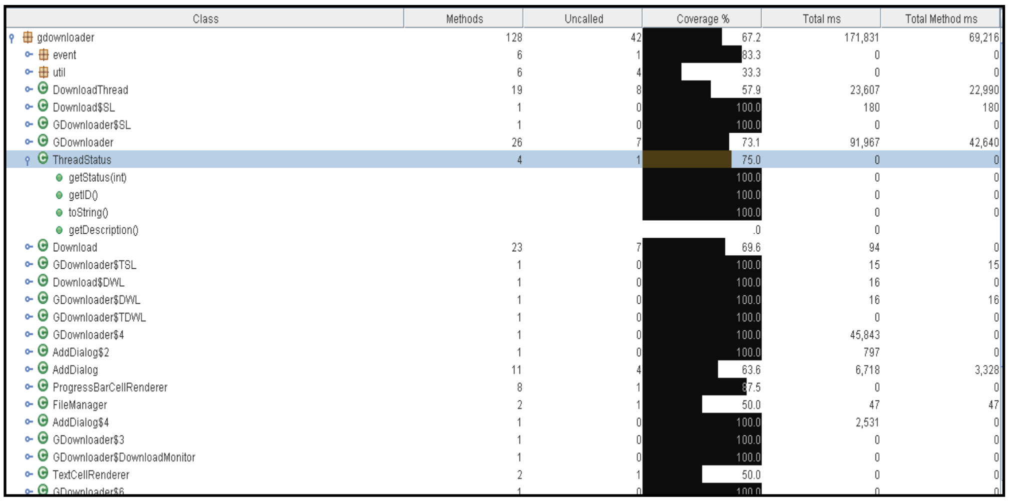


```
Root
  run - 1,546.73ms
    changeStatusTo
    notifyDataWriting
    getID
  actionPerformed - 45,843ms
    access$900
  valueChanged - 0ms
    onDownloadSelectionChange
  main - 172ms
    initMyComponents
    initComponents
  run - 0ms
  run - 0ms
  getTableCellRendererComponent - 0ms
  actionPerformed - 0ms
    access$1100
  windowClosing - never exited
  isCellEditable - 0ms
  isOpaque - 0ms
  getTableCellRendererComponent - 0ms
  validate - 0ms
  firePropertyChange - 0ms
```

initMyComponents()

Class	Method	Signature	Exits	Errors	Concurrent Threads	Total ms	Total Method ms	Average Method ms	% of Parent	% of Root
GDownloader	initMyComponents		1	0	1	15	0	0.00	8.7	8.7
GDownloader	main	String[]	1	0	1	172	78	78.00	.0	100.0

TABLE 3
Sample coverage analysis reported generated



Class	Methods	Uncalled	Coverage %	Total ms	Total Method ms
gdownloader	128	42	67.2	171,831	69,216
event	6	1	83.3	0	0
util	6	4	33.3	0	0
DownloadThread	19	8	57.9	23,607	22,990
Download$SL	1	0	100.0	180	180
GDownloader$SL	1	0	100.0	0	0
GDownloader	26	7	73.1	91,967	42,640
ThreadStatus	4	1	75.0	0	0
getStatus(int)			100.0	0	0
getID()			100.0	0	0
toString()			100.0	0	0
getDescription()			.0	0	
Download	23	7	69.6	94	0
GDownloader$TSL	1	0	100.0	15	15
Download$DWL	1	0	100.0	16	0
GDownloader$DWL	1	0	100.0	16	16
GDownloader$TDWL	1	0	100.0	0	0
GDownloader$4	1	0	100.0	45,843	0
AddDialog$2	1	0	100.0	797	0
AddDialog	11	4	63.6	6,718	3,328
ProgressBarCellRenderer	8	1	87.5	0	0
FileManager	2	1	50.0	47	47
AddDialog$4	1	0	100.0	2,531	0
GDownloader$3	1	0	100.0	0	0
GDownloader$DownloadMonitor	1	0	100.0	0	0
TextCellRenderer	2	1	50.0	0	0
GDownloader$6	1	0	100.0	0	0

2) Code Coverage: To value our tool's code reporting, we compare dangerous loadings perceive by the static and powerful analyses. In specific, we detected unsafe component loadings with the present dynamic technique [12] and evaluate its outcomes with our semi-automatic detection. In this assessment, we emphasis on application-level runtime unsafe loadings as load time reliant parts are filled by JRE-level code. We see that our semi-automatic model can find not only mainly of the dynamically-detected insecure loadings other than also several other potential ones also. We next offer a closer examination of the results.

Static-only Cases: Our static analysis notice many additional potential unsafe loadings. It's essential to understand whether they show actual mistakes or not. We physically studied these extra detected unsafe loadings to appraise the precision of our investigation. Specifically, we examined whether they are reachable from the admittance points of the programs, I.e., whether there survive paths from the access points to the identify sites of the insecure loadings in the plans' inter-method Call flow graphs (Inter-method describe flow graphs).

Note that these loadings marked as "Unknown" may still be obtainable because it is difficult to work out circuitous jumps in binary code, so specific manage flow edges may be lacking from the Inter-method call flow graphs. All the statically accessible unsafe loadings cause component load hijacking if the corresponding call sites are raise and also the target components have not been loaded yet.
External Parameters: A target pattern may be defined by a limitation of an exported function, which isn't invoked. One may offset this problem by examining the data flow dependencies between the dependent parts. Because the exported functions are frequently not appeal to by the parts, however, such an investigation does not assure to get all the objective specifications.
Unknown Semantics of System Calls: Comprehensive semantics of classification calls is frequently not documented, and at times also their names are not disclosed. We cannot examine nor copy them, when we experience such system calls. When details of such method calls become accessible, we may certainly add analysis support for them.

V. Related Work

We survey additional related work aside from the one on recognition of dangerous loadings [12], which we have already discussed. Our approach performs static examination of binaries. Within this setting, assessment Set Analysis (VSA) [2, 18] is probably the mainly closely associated to ours. It combines numeric and indicator analyses to calculate an over approximation of numerical values of program variables. Evaluate to VSA, our technique focuses on the calculation of string variables. It is also, demand- driven and uses context-sensitive emulation to level to real-world substantial applications. Emblematic analysis [11] may be utilized to calculate values of the program factors, once we discussed previously, instead of emulation. However, symbolic techniques normally suffer starting poor scalability, and more importantly, it's not practical to symbolically cause about method calls, which are often quite complex. Our new use of context susceptible emulation provides a useful solution for dispensation the ideals of program variables. Starting with Weiser's seminal work [25], program segmenting has been extensively studied [23, 26]. Our perform is associated with the large body of effort on static segmenting, in particular the SDG-established techniques. Standard SDG-based static segmenting techniques [1, 5, 9, 16, 20, 21] build the entire SDGs beforehand. In contrast, we build control - and data - flow dependence in sequence in a fashion, beginning with the specified segmenting criteria. Our segmenting technique is also, modular because we model each call site utilizing its callee's inferred outline that abstracts absent the internal addiction of the callee. In particular, we handle a phone as a non-branching training and approximate its dependencies with the callee's synopsis info. This optimization tolerates us to conceptual away detailed data flow dependencies of a purpose using its equivalent call instruction. We make an successful trade-off amid accurate and scalability. As shown by our evaluation results, function prototype information may be efficiently computed and give exact results for our location.

Our segmenting algorithm is demand driven, and is hence also connected to demand-driven dataflow analyses [10, 17], which have been projected to enhance investigation performance when entire dataflow facts are not needed. These strategies are similar to ours because they also leverage caller-callee affiliation to rule out infeasible dataflow paths. The principal distinction is that we use a straightforward prototype analysis to construct concise function summaries as a substitute of directly crossing the functions' Intra procedural dependence graphs, I.e., their PDGs. Another difference is the fact that we generate context sensitive executable program obstruct for emulation to prevent the problem in thinking about method calls.

VI. Conclusion And Future Work

We've presented a semi mechanized DSU evaluation approach to discover insecure loadings. The core of our evaluation is techniques to just and scalable to extract which parts are loaded at a specific consignment call site. We released a java stack log extraction and evaluation procedure, which combines modular and incremental slice construction with the emulation of call flow associated blocks. Our evaluation on nine admired Windows submission shows the effectiveness of our technique. Due to its good scalability, precision, and protection, our approach serves as an effective balance to dynamic detection [12]. For potential work, we'd want to think two interesting directions. Since unsafe loading is a general concern as well as relevant to additional runtime locations, consequently we intend to extend our technique and assess unsafe part loadings in additional run time environments including CLR. Second, we plan to investigate how our approach can be improved to cut back emulation failures

References

[1] Akos Kiss, J. Jasz, G. Lehotai, andT. Gyimothy. Interprocedural static segmenting of binary executables. In *Proc. SCAM*, 2003.
[2] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. CC*, 2004.
[3] G. Balakrishnan and T. Reps. Divine: discovering variables in executables. In *Proc. VMCAI*, 2007.
[4] G. Balakrishnan and T. Reps. Analyzing stripped device-driver executables. In *Proc. TACAS*, 2008.
[5] D. Binkley. Precise executable interprocedural blocks. *ACM Lett. Program. Lang. Syst.*, 2(1-4):31-45, 1993. Dlopen man page. http://linux.die.net/man/3/dlopen.
[6] Dynamic-Link Library Search Order. http://msdn.microsoft. Com/en-us/library/ms682586 (VS.85).aspx.
[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319- 349, 1987.
[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural segmenting using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26-60, 1990.
[9] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proc. FSE*, 1995.
[10] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385-394, 1976.
[11] T. Kwon and Z. Su. Automatic detection of unsafe component loadings. In *Proc. ISSTA*, 2010.
[12] J. Lim, A. Lal, and T. Reps. Symbolic analysis via semantic reinterpretation. In *Proc. SPIN*, 2009.
[13] J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *Proc. CC*, 2008.
[14] Microsoft Portable Executable and Common Object File Format Specification. http://www.microsoft.com/whdc/system/platform/ firmware/PECOFF.mspx.
[15] A. Orso, S. Sinha, and M. J. Harrold. Incremental segmenting based on data-dependence types. In *Proc. ICSM*, 2001.
[16] T. Reps. Solving demand versions of interprocedural analysis problems. In *Proc. CC*, 1994.
[17] T. Reps and G. Balakrishnan. Improved memory-access analysis for x86 executables. In *Proc. CC*, 2008.

[18] T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. A next-generation platform for analyzing executables. In *Proc. APLAS*, 2005.

[19] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up segmenting. In *Proc. FSE*, 1994.

[20] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph- based segmenting of programs with arbitrary interprocedural control flow. In *Proc. ICSE,* 1999.

[21] 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.

[22] F. Tip. A survey of program segmenting techniques. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1994.

[23] Types of Dependencies. http://dependencywalker.com/help/ html/dependency_types.htm.

[24] M. Weiser. Program segmenting. In *Proc. ICSE*, 1981.

[25] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program segmenting. *SIGSOFT Softw. Eng. Notes,* 30(2):1-36, 2005.