RC24979 (W1004-059) April 16, 2010
Computer Science

IBM Research Report

Discovery of Hard-coded External Dependencies in Enterprise Production Environments

Nikolai Joukov[1], Vasily Tarasov[2], Birgit Pfitzmann[1], Sergej Chicherin[3], Marco Pistoia[1], Takaaki Tateishi[4]

[1]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598 USA

[2]Computer Science Department
Stony Brook University
New York USA

[3]IBM Russian Systems and Technology Laboratory
Moscow, Russia

[4]IBM Tokyo Research Laboratory
Tokyo, Japan

Discovery of Hard-coded External Dependencies in Enterprise Production Environments

Nikolai Joukov[1], Vasily Tarasov[2], Birgit Pfitzmann[1],
Sergej Chicherin[3], Marco Pistoia[1], and Takaaki Tateishi[4]

[1]IBM T. J. Watson Research Center, Hawthorne, NY USA,
[2]Computer Science Department, Stony Brook University, NY USA,
[3]IBM Russian Systems and Technology Laboratory, Moscow, Russia,
[4]IBM Tokyo Research Laboratory, Tokyo, Japan

ABSTRACT

Many enterprises perform data-center transformations, consolidations, and migrations to reduce costs and make IT greener. These projects start with discovery of infrastructure and applications and their dependencies. Typically, this is done by network monitoring and middleware configuration analysis. However, certain dependencies may not be detected without code analysis.

We designed and implemented the first code-analysis technique for discovering hard-coded dependencies, based on static string analysis. Key novel aspects include the ability to localize the code and associated files in a production enterprise environment, an analysis for identifying functions that can access external resources, and a program-environment analysis (linked to the string analysis) for inferring values originating outside of the program. The approach is sound under reasonable assumptions about the underlying components.

We analyzed 1097 Java EE applications from three enterprise environments. The vast majority had hard-coded dependencies that required our novel analysis techniques. Such applications need special treatment in transformation projects.

Categories and Subject Descriptors

D.2.9 [**Software**]: Software Management—*Software maintenance*

General Terms

IT Services

Keywords

discovery, IT services, code analysis

1. INTRODUCTION

Data-center transformation, IT optimization, consolidation, green projects, virtualization, and migration to cloud are some of the buzzwords under which major enterprises are currently undertaking projects to make their IT infrastructures and operations more efficient. The economic climate has actually increased the investment in such projects.

Virtually every large-scale transformation project starts with IT discovery, because enterprises almost never have detailed, comprehensive, and up-to-date information about their IT infrastructure. One reason is that enterprise infrastructures have grown over decades, with new-generation technologies—such as Service Oriented Architecture (SOA)—added, but old-generation technologies never pushed completely out. Another reason is mergers and acquisitions, where different infrastructures are mixed. A third reason is that usually documentation is either out of date or in silos for certain departments. Similarly, short-term problem-solving tasks often need discovery, because the IT components of problematic applications are not always sufficiently understood.

For transformational projects, dependencies between different software components, e.g., from a Java EE application to a database, play a crucial role. Components that communicate with each other typically need to be treated together, so that all business applications continue to work after the transformation. Dependencies across different servers are of particular interest, but also dependencies between separate applications on one server. Figure 1 shows some typical dependencies. An arrow from Component A to Component B means that A depends on B.

Discovery of application details and dependencies in production

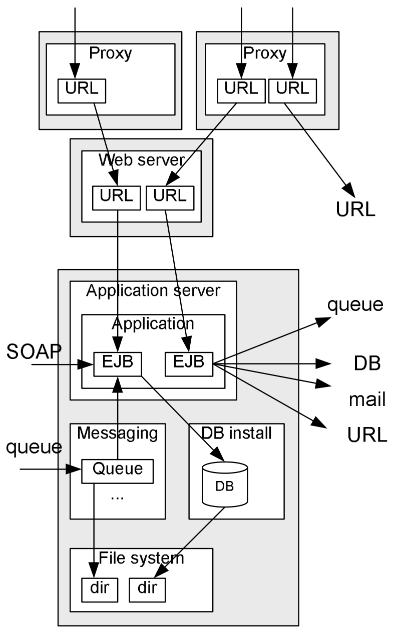

Figure 1: Dependencies between software components. Grey boxes represent servers.

environments is still a considerable challenge. Existing tools do not provide all the information necessary or are very hard to deploy. Therefore, it is still quite common to perform major parts of the discovery process manually, which causes significant delays, errors and costs. Many potential enterprise IT optimizations are never undertaken because of these problems. Any additional automation in dependency discovery can therefore have huge effects beyond the direct savings. In this paper, we are particularly interested in dependencies that are wholly or partially embedded in code, e.g., in the EJBs in Figure 1. We call these dependencies *hard-coded*.

In enterprise situations, we may immediately get an objection: Java code in enterprises mostly exists in Java EE environments, and the Java EE specification prescribes that all used resources are declared in standard resource files. So we would only need to analyze these resource files, which is much simpler than analyzing the code itself. However, while this is what we described in prior work and still use a lot, we started to analyze semi-manually whether this specification is really followed. Surprisingly, we saw many exceptions, i.e., dependencies directly in the code without corresponding resource descriptions. We call these hard-coded dependencies, even if some of them use certain non-Java-EE standard configuration files. This led us to develop automated techniques to analyze hard-coded dependencies, which we describe here. The evaluation in Section 6.1 shows how prevalent hard-coded dependencies are.

Our analysis is static, and based on underlying techniques for discovery in production environments, here Galapagos [19], and for code analysis including static string analysis, here WALA-SA [11]. We use the string analysis in determining the actual resource names, which are typically strings. Major novel aspects beyond the combination of these two ingredients are the following:

1. The ability to localize the code and associated files (such as an XML configuration file) in a production enterprise environment.

2. Sound identification of the basic constructs that allow code to access external resources.

3. Abstract interpretation of configuration repositories and variables from the program environment, such as command-line inputs and paths, that may be used in the derivation of resource names. This component is integrated with the string analysis.

Without the third novel aspect above, the string analysis for resource names often conservatively returns `*` (a wildcard), or a concatenation where an important part is still `*`. We also show statistics of that in Section 6.1. Note that the additional configuration repositories mentioned above are not the same as the configuration files in a Java-EE-compliant scenario, because their names and locations are not standardized, nor are the actual field names, even when standard Java property files are used. Hence taking these non-standard repositories into account is very important, but a significant analysis challenge too.

Furthermore, the third aspect leads to recursion: The additional configuration repositories are also external resources; determining their names requires the same approach and can depend on yet more external configuration repositories.

Our approach is largely independent of whether Java code runs on a Java EE server at all.

The remainder of this paper is organized as follows: We start with the problem setting and examples in Section 2, and review related work in Section 3. We present the design of our algorithm for discovering hard-coded dependencies in Section 4. We describe our implementation for Java EE applications running on IBM WebSphere in Section 5. In Section 6, we evaluate our approach: First, we present statistics about the occurrence of different types of dependencies, which need more or less sophisticated analysis techniques, in over a thousand applications from real enterprise environments. Secondly, we demonstrate, under reasonable assumptions, that our approach is sound in that it finds all external dependencies with possible over-approximation. Thirdly, we discuss one of these assumptions in more detail. Finally, we present performance results. We conclude in Section 7.

2. PROBLEM SETTING AND EXAMPLES

Our goal is to discover the external resources that a program may use, such as files, databases, messaging queues, and network resources accessed via URLs. In this section, we first present the discovery setting in more detail, then show some code samples of increasingly complex dependencies on external resources, which our algorithm has to handle, and finally discuss our goals with respect to soundness and precision.

2.1 Discovery Scenario

Figure 2 shows a program with its environment. The external resources that we are primarily interested in are shown in bold on the right. They may or may not reside on the same server or image. The program has a basic run-time environment, such as a Java EE application server, and may depend on other code libraries. Important aspects of how the program runs come from the command-line input that the program is started with, and it may have configuration files or a configuration database; we summarize this as the *configuration repository*. Furthermore, full resource names may depend on current paths.

In our discovery scenario, in contrast to typical code analysis performed during product development or test, we consider programs together with an environment. For instance, we may have a long-running banking or hospital application. Significant parts of the environment, such as configuration repositories and the initial command-line input, are static at this time. Even if an enterprise production program does not run continuously, it is typically restarted each time by a fixed start-up script, and thus with some fixed resources. Technically, configuration repositories are also external resources, but semantically they play a secondary or metadata role, and often contain information about primary external resources.

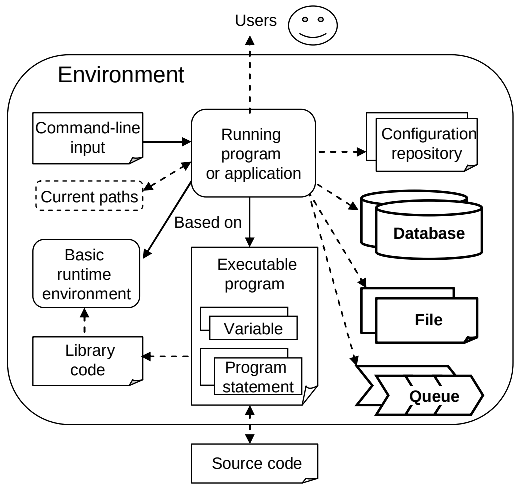

Figure 2: A program with its environment

Furthermore, in a discovery scenario, source code is not always available. Our analysis therefore has to work with binary code, which is fortunately not a big problem with Java.

2.2 Examples

Figure 3 shows three examples of dependencies of a Java program on a file. The stop function in all three cases is a constructor of the class `FileInputStream`. The name of the file represented by `f0` is given as a constant string. However, we will see in Section 6 that this is not a very common case; hence we do need analysis techniques that can handle more complex cases. The name of the file represented by `f1` is obtained by concatenating a constant string and the value of variable `name`. This value depends on another variable, `version`. Naively overapproximating the value of `f1` as `/data/*` would be sound, but too general. However, string analysis, which is part of our approach, can compute the two possible filename values as `/data/matrix.old` and `/data/matrix)`. The name of the file represented by `f2` is taken from the list of the program's command-line arguments.

```
FileInputStream f0, f1, f2;
f0 = new FileInputStream("/data/matrix");
if (version == "old")
   name = "matrix.old";
else
   name = "matrix";
f1 = new FileInputStream("/data/" + name);
f2 = new FileInputStream(argv[1]);
```

Figure 3: Three types of dependencies of a program on files

Figure 4 shows a more complex, but not uncommon, example of how a resource name may depend on configuration parameters outside the code. All four program lines may be in different parts of the program; in particular, the first two may be in one class and the second two in another class. In this example, `DriverManager.getConnection` is the stop function, and its first parameter is the database URL, which is derived as a concatenation of two strings. As the value of the variable `db` is unknown in the code alone, existing string analysis would overapproximate the URL as `jdbc:db2://*`. This is no real help in identifying the database. Our extension of string analysis to definitions outside the code allows us to go further: It detects that the variable `db` is assigned a value only once in the code, namely `props.getProperty("db.dbname")`. The object `props` is also assigned a value only once: the return value of `getResourceAsStream("settings.properties")`. This is another stop function that refers to an external resource, so in general we have to use string analysis recursively. However, in this case, the resource name is a constant string, which tells us right away that the resource is the file `settings.properties`. With the addition of path analysis based on where the run-time environment will look for this file at this point in its execution, we find the actual file. Finally, we emulate the way the run-time environment searches for a property in such property files, which is standard in Java. For the example property file in Figure 5, the emulation finds that the value of variable `db` can only be `sales`. Substituting this into the results of the initial string analysis yields that the database accessed in this program is precisely `jdbc:db2://sales`. Figure 6 shows an XML property file similar to the Java property file of Figure 5. Java programs often use custom XML properties files too, but they are less standardized and thus require heuristics in the analysis; if those do not succeed we still have to use `*` as their output for soundness, and potentially add manual analysis based on the component origins that the WALA-SA string analysis gives us.

```
Properties props = new Properties();
props.load(getClass().getResourceAsStream(
    "settings.properties"));
String db = props.getProperty("db.dbname");
Connection cn = DriverManager.getConnection(
    "jdbc:db2://" + db, "admin", "pwd");
```

Figure 4: Dependency on a database via properties

```
# DATABASE configuration
db.dbname=sales
db.maxcon=5
db.mincon=2
```

Figure 5: Java property file `settings.properties`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java...
<properties>
  <comment>DATABASE configuration</comment>
  <entry key="db.dbname">sales</entry>
  <entry key="db.maxcon">5</entry>
  <entry key="db.mincon">2</entry>
</properties>
```

Figure 6: Java XML property file

2.3 Soundness and Precision Goals

Discovering precisely the resources that a program will use is undecidable. Fortunately, for most use cases involving IT transformations, optimizations, or problem solving, a reasonable overapproximation is very useful. The only requirement in most cases is for the analysis to be sound, which means that it will capture all the dependencies. For example, assume that, for backup purposes, we want to derive a list of files a program depends on. If we can only derive that all these files are in a certain directory (for example, all the filenames have the pattern `/usr2/bak/*`), it is safer to back up all files in that directory than none, even if only one file may actually be used. Therefore, our goal is to derive all resource dependencies with reasonable overapproximation if necessary. In Section 6.2 we will discuss under what (reasonable) assumptions we achieve this goal; we will also mention a few cases where it is useful to deviate from soundness, at least for the time being.

A key observation for this soundness is that, at least in a type-safe language such as Java, to access an external resource, a program ultimately has to invoke a function out of a limited set of functions defined in the run-time environment. We call a function that is defined outside the program and gives access to an external resource a *stop function*. Parameters to stop functions include the names, types, and sometimes locations of the relevant external resources.

For our analysis, we can treat libraries either as part of the program or as an extension of the basic run-time environment. We usually do the former for customer libraries and the latter for well-known standard libraries that we expect to encounter more than once. We call every function defined outside of the program an *external function*.

3. BACKGROUND AND RELATED WORK

Today IT asset and dependencies discovery tools are available from every major IT services vendor. Such discovery tools probe network nodes with requests [2, 9], monitor network traffic [6, 10, 16, 17, 29], or analyze software configurations [1, 4, 13, 19, 24].

None of the tools that analyze software configurations touches code, i.e., analyzing code in this context is one of the novelties of our current paper. The tools only analyze packaged middleware and applications such as databases, Java EE servers, and ERM systems. For Java EE servers, they only analyze objects and relations explicitly configured at the server level, such as what EJBs are deployed in it and what resources they declared in the standard way. Typically, software configuration analysis is done by interacting with running software via its management interfaces. In the Galapagos tool we switched to analyzing configuration files directly (mostly after the publications [18, 19]), because this is often possible with fewer privileges, such as without the password of the administrator of a specific application server or database. This approach also facilitates building up the novel code analysis for production servers where we have to automatically find the code for analysis first, because the management interfaces would not give us the code and all related files.

In addition, Galapagos uses a script-based approach where everything that needs to be done on the production servers is done by one rather short, non-interactive script, while the bulk of the analysis is done on a back-end analysis server. This approach was started in [14]. The advantage is that the system administrators of the production servers can run the script without giving any account to the discovery team or its software, and can first validate that the script indeed makes no changes on the production servers, reads no customer data, and has low resource consumption. This can drastically reduce the approval time and thus the real-life overall discovery time, compared with approaches where a central server logs into the production servers, for which it needs credentials and where the system administrators have less control over what happens. In situations where such logins are possible, it is easy to push the script out automatically, so there is no downside to this approach in terms of deployments. In the following, we use this approach also for the code analysis. Figure 7 shows a small Galapagos example, here for an IBM DB2 installation for which configuration information was captured and parsed. We see that DB2 is installed in `/opt/ibm/db2`, and two instances are configured with user home directories in `/home`. DBL is a local database, while DBR is the remote definition of a database located on `server1`. Next, this result is connected with other discovered software components. In particular, if discovery was also run on server `server1`, the link to DBR is connected to the model of DBR discovered there, potentially using alias analysis. Similarly, in this paper we want to link a potential hard-coded dependency from a Java application to, say, DBL, to the model of this database.

Monitoring can generally complement static discovery, because they have opposite pros and cons: The main benefit of monitoring is that it is relatively product-neutral. For instance, one sees a connection in a `netstat` table or a NetFlow result independent of what middleware or application made it. The main benefit of configuration analysis, or rather any static analysis including our new code analysis, is that one can also see components and connections that are not active during the monitoring time, e.g., that only occur at certain times of the year or only in exceptional situations. Often one even only gets a one-shot monitoring chance, e.g., `netstat` information at one point in time, which is clearly very incomplete. Hence most commercial tools combine both approaches [4, 13, 22]. In our specific use case of analyzing hard-coded dependencies in Java programs, monitoring cannot help much: First, we are also interested in dependencies within one server, such as the dependencies on a messaging queue and a database shown in Figure 1. A network-based tool cannot see these. And while monitoring of software connections inside servers is possible in principle [5], current production servers in enterprises almost never have such instrumentation, and it is totally impossible in practice to get authorization to deploy OS instrumentation on existing servers that run production-level business applications, except sometimes the `lsof` tool. Secondly, for many optimization use cases we are in fact interested in determining *where* the dependencies are configured because something may need to be changed about them. Static analysis can find this precisely, in terms of detailed components such as databases and in terms of the actual configuration or code line, while monitoring (both network-based and operating-system based) usually only finds it at the overall process level.

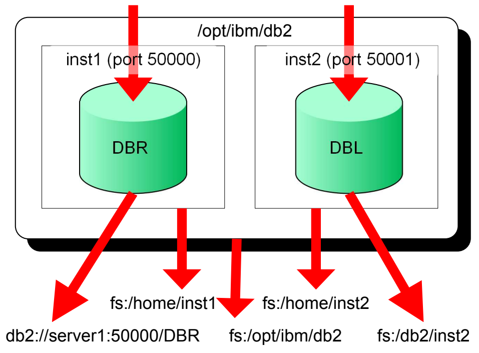

Figure 7: Example of existing Galapagos configuration-based discovery

Let us now look at the code analysis background. Finding dependencies within a body of code is standard (call graphs etc.). However, we are interested in external dependencies, i.e., dependencies on anything outside the current body of code. Furthermore, finding the code on production servers in the first place is part of our task.

String analysis for Java, as we will use as a basis, was introduced in the Java String Analyzer (JSA) by Christensen and others [7, 15]. It can be seen as an instantiation of abstract interpretation [21] for string variables. JSA first builds a program's control flow graph and then extracts the grammar for the variable in question. In the grammar, each string expression is represented as a non-terminal. At the second stage, JSA substitutes each non-terminal by a context-free grammar that emulates the result of the corresponding string operation. Its authors used the Soot framework [23, 25] as a front-end for converting Java code to an intermediate representation.

String analysis is an important instrument for validating dynamically generated content. Applications often construct SQL queries dynamically at runtime and they do little or nothing to check that the resulting query is valid. Gould et al. addressed this problem by applying static string analysis for Java to derive possible values of SQL queries [12]. Minamide applied JSA to approximate the contents of dynamically generated PHP pages [20]. The approximation is useful for validating the pages and detecting potential vulnerabilities in cross-site scripting. String analysis was also used for security-related code and policies verification by Wasserman and Su [27, 28].

Geay et al. presented a string-sensitive permission analysis for Java and Common Language Runtime (CLR) applications [11]. They adapted string analysis to disambiguate permission propagation paths. The underlying string-analysis algorithm is based on Minamide's work, but introduces a novel labeling feature, which allows tracking the origin of each component in the resulting string. These string analysis extensions were added to IBM's Watson Libraries for Analysis (WALA) [26]; we call this WALA-SA. As we are also interested in tracking the origin of string components, e.g., for potential later changes, WALA-SA was a natural choice as the underlying static-analysis engine for our current paper.

Christodorescu et al. [8] demonstrated the possibility to deduct string values not only for information-rich code, such as Java bytecode, but also for dense x86 object code. The main problem solved by the authors is the extraction of sufficient information from the object code to reconstruct string operations. This may be interesting for future extensions of our analysis of hard-coded dependencies to x86 object code.

A common alternative to a full-fledged string analysis is the analysis based on constant propagation, where all non-constant values are approximated with `*`, denoting an arbitrary string. Due to this simplification, constant propagation analysis runs faster. However, the locators of external data often undergo modifications in the program (typically concatenations). Hence an analysis solely based on constant propagation would not obtain enough precise results for external dependencies. In their work, Geay, et al. reported a precision improvement of 53% when using WALA-SA to compute resource names, as opposed to using simple string-constant propagation.

We are not aware of prior work of analyzing code together with parameters from its runtime environment, as we will present it below. In fact, many other use cases analyze code during its design and test phases and before it is deployed. At this point, the environment is not yet given. In this sense, optimization and transformation tasks for running enterprise applications are special.

4. DESIGN

In this section, we describe our algorithm, which generalizes from the examples in Section 2.2. We start with an overview, and the following subsections give more details about each step.

Recall that one of our goals is to concentrate everything that needs to be performed on the production servers (where the actual application is running) into a short script, and to do as much as possible on a dedicated discovery server. Hence we fetch the code and certain environment elements from the production servers with the script, and perform all the analysis on our own server.

So far we only mentioned strings as the objects for which we need abstract interpretation, because resource names are typically strings, as we saw in Section 2.2. However, strings are not the only objects involved. For example, the Java `Properties` class that we also saw in Section 2.2 extends the `HashTable` class. Fortunately, string analysis, at least in WALA-SA, can be applied to non-string objects by by artificially defining these objects to be of string types and by writing emulators for functions on these objects. By emulators we mean string-equivalents of these functions in a format that can be applied when the static string analysis comes across such a function. In our context we call this configuration analysis, because it mainly concerns the configuration repositories in the sense of Figure 2. Thus, the entire abstract interpretation can run in a uniform way for the actual string variables defining external resource names, and for non-string property variables from which input strings might be fetched. This yields the following algorithm structure.

1. *Code and configuration gathering.* Detect and collect program modules, configuration, and state data, and transfer collected information to the discovery server.
2. *Stop function identification.* Detect all the stop functions in the given program. We do this by matching external functions used in the program against a database of external functions that we have previously classified into stop functions or non-stop functions, and by newly classifying unrecognized functions. The classification includes which parameters of the stop functions actually denote an external resource.
3. *String and configuration analysis.* Perform string analysis to derive an abstract interpretation of parameters of stop functions that define external resources. This includes the configuration analysis for non-string classes such as `Properties`, with our own abstract emulators for their methods.
4. *Locating and Loading Configuration Repositories.* If one of the emulators requires loading of a named external resource, such as a command-line argument or a configuration repository, locate that resource by determining the correct path to it and possibly by refining wildcards by comparison with actually existing resources. Then load it from the local copy produced in Step 1, and continue the emulation.
5. *Refetching.* If the located repository was not collected from the production servers in Step 1, collect it before resuming the loading and thus the emulation. One will typically try to analyze all programs from one server before refetching, because refetching typically involves interaction with the server owners.
6. *Postprocessing for overall discovery*. When we have a result for one of the initially desired external resources, i.e., the stop function parameters identified in Step 2, we link it to the model of the external resource in the overall discovery result.

In the following, we describe each of these steps in more detail.

4.1 Code and Configuration Gathering

Before analyzing a program, we have to detect it and fetch its code and related state and configuration data. Recall that we mainly consider situations where we have to analyze a running production environment where very little is known in advance.

Programs can be detected based on the currently running processes, registered packages, standard installation paths, disk scanning if audit and performance constraints on the production servers permit it, and known processes that start others. For example, the `inetd` daemon can start a program upon receiving a network request. As none of these techniques is perfect, one needs a good combination of them if the goal is to find all custom programs on a server, and we do not claim soundness for this step. In our specific situation, we only have to gather all applications officially declared in certain application servers, so we do achieve soundness for our more restricted situation.

In general, the configuration files used by a software component can be in arbitrary places on a disk. Fortunately, in practice there are heuristics that allow us to locate and fetch configuration files in advance. For example, Java EE applications usually have configuration files located inside their Enterprise Archive (EAR) files, or somewhere in the `classpath` environment variable. Hence by fetching EAR files with the program modules, and files from the `classpath`, we obtain most of the related configuration files. The others are fetched after the initial code analysis in Step 5.

In addition to code and configuration files, we fetch other system parameters defining the environment of the program being analyzed: the directory it is started in, environment variables, and command-line arguments either from `ps` output if the program is currently running, or from a parent script. Most of this is not done per program, but once for each server, e.g., by executing the `ps` command and transferring its entire output back. We also fetch the configuration of the underlying application servers where our Java code runs.

4.2 Stop Function Identification

Recall that a key observation for our analysis is that in order to address an external resource, a type-safe program must ultimately call a stop function. A run-time environment may contain a huge number of functions that a program could potentially call, and Java EE run-time environments vary even over different versions of the same application-server product. It would be hard to classify all those function in advance into stop functions and non-stop functions. Instead, we take the following approach:

- Given a program, we build a callgraph of the functions that are reachable from the program entrypoints. By entrypoints we mean functions where program execution may start, such as the `main` method of an application. Java EE applications have multiple entrypoints. Note that the information about initial calls that we obtained from running processes and startup scripts in Section 4.1 contains calls to the run-time environment, which forwards them to entrypoints of the custom code. The callgraph gives us a set F of functions that may actually be called. This process eliminates a large number of external functions (some of which are stop functions) from the analysis. This way, we also do not list external resources that are specified in the program but are never used.
- We detect which of the functions in F are defined inside the given program; all the others we call the *external functions* of the program. These external functions form a set E.
- We maintain a database D of external functions that we manually classified into stop functions or non-stop functions. More precisely, "external" is a notion relative to a program, but as long as we have unambiguous function identifiers, it does not matter whether a function that is external for one program is not external for another. For instance, the `Vector.add` constructor is not and `FileInputStream.<init>` is a stop function. As we will show in Section 6.3, this database converges at a reasonable rate. For known stop functions, we also maintain a list of parameters that define external resources. E.g., `FileInputStream.<init>`'s first parameter defines the file name.
- We look up all functions in E in the database D and obtain a subset S of known stop functions, as well as a set U of previously unknown external functions used in our program. We newly classify these functions, add them to the database, and add those that are stop functions to the set S for this program. Classification of new functions is, for now, done manually. We also look up the parameters defining external resources for the functions in set S. Thus we have a starting set of parameters in specific calls inside the callgraph for which we want to perform the remaining steps.

4.3 String and Configuration Analysis

External resource names or addresses are typically expressed as `String` parameters of stop functions. Even where this is not directly the case, for example, with the `URL` class in Java, the class is typically based on strings or can be reinterpreted as strings. Hence, we use string analysis to obtain an abstract interpretation of the possible values of these parameters, in the form of a context-free grammar (CFG) or regular expression. String-analysis engines emulate string operations. They normally model basic operations such as string concatenation, and emulate externally defined string functions as transducers [20]. String-analysis engines are available for a variety of languages. We use WALA-SA, the string analysis engine based on the Watson Libraries for Analysis [11]. WALA [26] has configurable front-ends for various languages. We focused our analysis on Java EE bytecode applications, because we usually do not have access to the sources of the programs we analyze. WALA-SA can output not only a grammar with wildcards, but an abstract interpretation that shows the originating program points leading to these wildcards, such as the assignment to variable `db` in Figure 4, if an emulator for the `Properties` methods did not exist yet. This is important for constantly improving our analysis to detect new methods that we need to emulate.

Figure 3 and 4 showed examples where a file name depends on a command-line argument, and a database name is derived from a configuration file. These cases are common, and classical string analysis would return wildcards for the corresponding string parts Hence we need abstract interpretation not only for strings, but for such configuration objects. We call this *configuration analysis*. For instance, for the code fragment in Figure 4, we provide emulators for the methods `load`, `getResourceAsStream`, and `getProperty`. In our case of using WALA-SA, we can piggyback this configuration analysis onto the string analysis, i.e., we only have to add configuration emulators. This works because WALA-SA enables us to declare classes as string-equivalent; for instance, we do this for the `Properties` class. The emulators treat configuration objects as actual strings at their interfaces, in order to be compatible with WALA-SA. For instance, the emulator of `getResourceAsStream` reads the properties file into a string variable `props`. The emulator of `getProperty` gets two strings as input (`props` and `db.dbname` in our example) and outputs another string (`sales` in our example). In the implementation of the emulator of `getProperty` we profit from the fact that the structure of Java properties files is standard, i.e., the emulator can parse it according to the few predefined ways of how name-value pairs can be represented in such a properties string.

For readers familiar with string analysis, let us mention that we do not define full string transducers for these functions. Transducers are the standard mechanisms to emulate string functions such as `append` and `substring` in WALA-SA, because transducers can be applied to CFGs, the representation of sets of possible strings in the abstract analysis. As we know that the real `Properties` class is only manipulated with the methods defined for it, we can derive that the corresponding emulated properties grammar is never represented by an arbitrary CFG, only a string or an explicit enumeration of a set of strings. Hence a simpler directly coded emulation is sufficient instead of a formal transducer.

Another important emulator derives command line arguments from the `ps` command output captured on the source server. This emulator knows the command line formats and argument numbering schemes on typical operating systems. It requires at least the program name as an input in order to find the correct line in the `ps` output. Furthermore, here we needed to implement additional Java EE support in WALA, because in Java EE, the external call as recorded in `ps` or in a startup call goes to the application server, which forwards it to an appropriate registered program (e.g., a servlet). These forwarding functions and their potential changes of the arguments have to be modeled in order to obtain what arguments the actual custom code obtains.

Generally, emulation of configuration files is not easy because they can have custom formats. We envision that even for some custom configuration files one can derive parameter values based on a set of heuristics, e.g., if a custom XML file contains key-value pairs in a format as in Figure 6; of course, heuristics that are not validated for a particular use case cannot be guaranteed to be sound. Fortunately at least for our primary use case, most configuration repositories are standard Java properties files; we show related real-word statistics in Section 6.1.

4.4 Locating and Loading Configuration Repositories

In Step 4 of the algorithm, we have to actually locate a configuration repository (e.g., a set of configuration files) of a program, based on the string parameter that the code uses to address it, as identified in Step 3. The challenge is that the name in the code is not always a complete address.

In simple cases, the string parameter itself contains all the information. For example, if `FileInputStream` is called with an absolute file path, this path uniquely identifies the file, and the file is on the same server. Similarly, if a resource name is a URL, the URL is typically a final result.

In other cases, the name or address that the code uses is relative, and we need to make it absolute. For instance, the Java runtime environment looks for properties files in the directories defined in its classpath. Hence this is also where our emulator looks for it. We retrieve the classpath from the `ps` output line for the given program on the given server, where it is visible as the `-classpath` option.

Other Java methods look for files relative to the current working directory. We use OS-specific methods to retrieve the initial current working directory of a process. However, the current working directory gets changed during program execution, and we need to know what values it might take at a particular point in the code analysis. Such context parameters can be treated similar to normal parameters in the string analysis, except that they are not explicitly defined. For example, by defining a global string variable CWD for the current working directory, and by emulating related functions (which are necessarily external), one can achieve that CWD is a parameter to all file-related stop functions. In this way, relatively defined file names can be converted to absolute file names by concatenating the CWD and the relative name at the same place in the program.

Once we have the full address of the configuration repository on the same or another server, we can compare it with the addresses of information prefetched during Step 1. (There may be alias resolution in this comparison.) We load it if it was prefetched.

If we do not have the full address from Step 4, but a reasonably restricted expression with wildcards, we can improve the precision of the result by matching this expression with our knowledge of real existing resources in the enterprise. For instance a dependency on a database `sales*` on a server `serv1.x.com` can be replaced with a dependency on database `sales3` on server `serv1.x.com` if discovery was also run on server `serv1.x.com` and this is the only database with such a name format there. In principle the same logic applies to file names, especially if read-only functions such as `getResourceAsStream` are used; however, as we typically do not fetch the entire directory structure with Galapagos (too resource-intensive) we can only really perform this matching if we are sure to have fetched all potential matches.

4.5 Refetching

If a configuration repository that has been located is not present, it needs to be fetched before the analysis can resume. As mentioned above, one should attempt to analyze all programs from one server before refetching (or even from all servers if remote configuration repositories are used) because even if the fetching is automated, the fact that one wants to perform it typically requires interaction with the server owner and is therefore time consuming.

4.6 Postprocessing for Overall Discovery

When we have a result for one of the stop function parameters identified in Step 2, i.e., a "primary" external resource of our program, we do not need to fetch it, but rather to link the models for the overall discovery result; e.g., the overall model now shows that a program depends on the database DBL from Figure 7.

If we do not have a model of that resource yet, we put a "placeholder" into the overall discovery result, representing everything that we learned about the resource from the dependency representation. This typically happens if discovery was not run on the other server (yet), or for files because we typically do not fetch the entire directory structures, or for resource types for which no (reliable) static discovery is available yet.

5. IMPLEMENTATION

In this section, we present a few more details of our particular implementation of the algorithm for discovering hard-coded dependencies described in Section 4.

For Steps 1 and 5, we build upon the Galapagos discovery tool. Galapagos in its current state, mostly added after the publications about it [14, 19], collects information with two script front-ends: one for a range of UNIX OSs and one for Windows. For our purposes, we made it fetch the installed applications in the IBM WebSphere and Oracle WebLogic application servers.

So far we focused on the analysis of Java EE applications deployed on IBM WebSphere application servers v.3 to v.7 because of our business needs. However, other recent Galapagos capabilities are to detect stand-alone applications that are either running at the time of discovery, or registered to run at regular intervals or based on certain events. Therefore, it would not be a huge effort to extend the analysis of hard-coded dependencies to Java applications that are stand-alone or deployed on other application servers.

Our implementation of Step 2 mainly uses the call-graph analysis features of WALA. We added the identification of external functions and their classification into stop functions and non-stop functions, as well as the database of already classified functions.

The WALA-SA string analysis [11] that we used in Step 3 runs on top of a 0-1-CFA context-insensitive callgraph constructed by WALA. Based on the callgraph, the flow of the strings in the program is deduced and constraint sets are generated [3]. By solving the resulting constraint system, the possible values of the variable in question are inferred.

Recall that as long as the resource name components are truly strings, and only transformed with standard string methods such as concatenation, we can simply use the existing WALA-SA. However, a large part of the actual operations on resource name components are not true string operations, but handle the loading and interpretation of configuration repositories, such as properties files, command-line arguments, and XML configuration files. Hence for these data types and operations, we had to provide our own emulators; we call this configuration analysis.

We wrote about two thousand lines of Java code to handle stop functions, improve WALA Java EE support, and add new method emulators. We also wrote about a thousand lines of shell script code for various tests and experiments. The Galapagos Windows and UNIX scripts that fetch WebSphere and WebLogic applications and configuration data are about 300 lines long combined. More than half of these lines are specifically written to fetch applications and related configuration files. We plan to release the WALA Java EE support improvements to the public domain.

6. EVALUATION

We examined three real-world enterprise environments, which we call `A`, `B`, and `C`. Environment `A` is the oldest; discovery was performed as part of its sunsetting. Discovery in Environments `B` and `C` was performed as part of optimization projects. Servers in

	A	B	C	Total
WAS installations	45	17	204	266
Application instances	111	104	5040	5255
Unique applications	56	49	1034	1097

Table 1: Statistics of the analyzed applications

environments A and C mostly run AIX, servers in B mostly Solaris OS, and a significant number of servers in all environments run Linux. During the discovery, we fetched all applications deployed on WebSphere application servers (WAS) and related system configuration information in addition to normal Galapagos information for the analysis of software configurations. Table 1 shows the total as well as per-environment statistics of the WAS installations we found in these environments, the application instances in them, and how many of these applications were unique. We ran our hard-coded dependency analysis on each unique application from our three environments.

6.1 Statistics of Hard-coded Dependencies

The most important result is that a large percentage of the Java EE applications in all three environments indeed have hard-coded dependencies. In other words, they contain dependencies that are either completely hard-coded in the application code, or specified in various non-Java-EE compliant configuration files. Thus they require the new discovery method we describe in this paper.

Table 2 shows the percentage of such application dependencies on messaging queues and databases, which are the kinds of dependencies that we are most interested in for our use cases related to IT transformation. Note that including a larger set of dependency types would only increase these numbers; we were conservative in particular to exclude non-code dependencies to other files inside the application archive files themselves. Still we see that in at least 30 to 90% of the applications we analyzed, static discovery with prior tools such as our own Galapagos tool and IBM Tivoli Application Dependency Manager (TADDM) Level 3 scans (the highest level) would miss some dependencies. Recall that whether monitoring (such as it exists in TADDM Level 2) would find them depends highly on the monitoring period, it would only work for cross-network dependencies or if at least `lsof` can be installed, and would not give the same level of detail, in particular about *where* the dependencies are configured in order to change something about them.

It is also interesting to observe that various enterprise environments seem to have different policies or guidelines about writing applications. In particular, in environments A and C most databases are handled according to the Java EE specification, while this is false for almost all messaging queue dependencies. In environment B this is quite different.

Our second question was whether the string analysis is really necessary. In particular, if string parameters were mostly provided as constants, e.g., as in `File("/data/data.xml")` rather than `File(filename)`, one would be able to derive the external resource names without the complex string analysis. Table 3 shows the percentage of such stop function invocations. As the number of such easy cases is low, string analysis is really necessary to extract hard-coded application dependencies on external resources.

	A	B	C
Messaging queues (%)	94	31	92
Databases (%)	7	25	5

Table 2: Non-standard external dependencies in our three environments

	A	B	C
Directly loaded string values (%)	8	10	7

Table 3: Calls to stop functions with constant parameters

Thirdly, we evaluated the importance of modeling parameters defined in the environment of the program (not counting Java-EE-compliant resource files). Table 4 shows the number of configuration files used by our applications. We can see again that there seemed to have been different coding standards and guidelines, which resulted in different usage patterns for configuration files. In particular, applications in the older set A do not rely on XML files, while the newer applications in B commonly do this.

	A	B	C
Properties	439	99	26,732
XML	0	13	29

Table 4: Numbers of property and XML files used by the applications

6.2 Soundness

We claimed above that our algorithm is sound, i.e., that it will only overapproximate resources, but not omit any. Let us demonstrate why this is true under reasonable assumptions.

First, the underlying Java runtime environment is type-safe, and therefore it is not possible for a program to access an external resource, e.g., by directly accessing system resources or by reading beyond its assigned memory. Any access outside its own virtual machine must be mediated by a function. (This includes calls to the Java Native Interface which allows calling other languages – in this case our Java analysis will go as far as identifying that other code is called; it is not responsible for analyzing further what downstream resources the other code may depend on). Hence we have shown that every external resource access requires use of functions defined outside the program itself.

Our Step 2 finds all these function calls in a program. For the following classification into stop functions and non-stop functions, we have to assume that the manual part is done correctly.

The string analysis in Step 3 is sound by the underlying work, in our case by the soundness of WALA-SA [11]. In addition, we have to assume here that we have programmed our own new emulators for configuration methods such as command-line arguments and property files in a sound way. For instance, this means that we have to take either the same decision as the Java runtime or to overapproximate in cases where several equally named properties are available and the runtime environment will choose one of them. Note that on enterprise servers, command-line arguments are also provided based on configuration files or start-up scripts and not manually. Therefore, we assume that command-line arguments are as constant as configuration files.

For Step 4, we have to assume that our resource location algorithm is correct with respect to what the Java runtime would do, or overapproximates it. For instance, it is no problem for soundness if we cannot find a property file at all, we return `*`. It would be a problem if we returned a different property file than the runtime would, in cases there are several that match the name pattern we have. Hence we have to assume that our analysis of current paths and priorities is correct with respect to the Java runtime, and where there is uncertainty we have to fetch and analyze all potential files. If we perform a comparison with existing resources, as in the example with the database `sales3` in Section 5, then for soundness we implicitly assume that the dependency is not dangling, e.g., that it does not point to a non-existing database `sales4`, and that the discovery of resources of the given type is complete. Hence it is a judgement call whether one performs this comparison for the benefit of greater precision, or omits it in order to retain soundness under weaker assumptions.

For Step 5, fetching additional configuration repositories or program components, the same argument as for Step 4 applies.

As to Step 1, we fetch all applications that are properly declared in a supported application server, so these are the input applications to which the subsequent soundness arguments apply. A similar statement would hold if we were given the code to analyze in another way. If we are missing program components or properties files in this step (although it occurs rarely in practice), then the following steps will automatically treat these as external resources and point them out. If they are needed in Step 4 to determine a component of the name of another resource, then either Step 5 will be able to fetch them, or that name component of the other resource will be output as `*`.

6.3 External Functions Classification

So far we manually classify external functions into stop and non-stop functions. The total number of external functions that an application can potentially use is large; we estimate that at least $100,000$ such functions are available for Java EE applications. It is virtually impossible to classify all of them into stop and non-stop functions manually. Fortunately, the number of functions that are commonly used is relatively small. Figure 8 shows how the number of encountered external functions grew with the number of applications we analyzed. The number of used external functions is higher for the old and diverse environment A, and lower for the newer environments B and C. Overall, the number of newly seen external functions declines reasonably well as the number of analyzed application increases.

Manual external functions classification is an ongoing process for us. So far we analyzed about $3,000$ external functions and found that about 2% of them are stop functions. For lack of time, we do sometimes show external dependencies that we already found in applications where we had no time yet to classify all the external functions into stop functions or non-stop functions. For instance, statistics that just warn about the commonality of hard-coded dependencies can only get more alarming with finding more stop functions. In such cases, we are outside soundness in the sense used before (finding a superset of the external resources), but on the safe side for the application. Or any result may be better than no result, e.g., because all discovered dependencies can already be treated in a transformation. To achieve good stop functions coverage for such cases, we added commonly known stop functions to

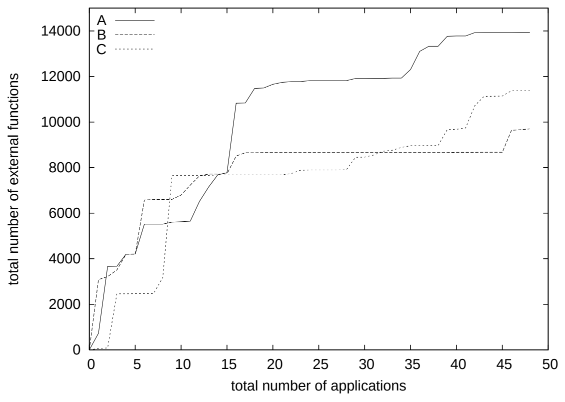

Figure 8: External Functions per Applications Analyzed

the list even if we did not encounter them yet.

For the future, we also consider moving the boundary of what we consider external functions and thus stop functions down, i.e., to analyze the Java EE runtime environments and other libraries used with our tool with respect to a more basic Java run-time environment.

6.4 Performance

Table 5 shows the performance of our hard-coded dependency extractor for several applications of different sizes. In this experiment, we ran the extractor for a single stop function that is invoked only once in the code. We conducted this experiment on a 2 GHz Linux machine with 4 GB of memory. For each application we list the number of classes in the program, the number of callgraph nodes, the number of production rules instantiated by the string analysis to evaluate the values of the string variables in the program, and the time required to perform the analysis. The applications are sorted by the number of classes in them, because this is a good approximation of program complexity. About two thirds of the execution time is spent on building the callgraph and doing other one-time per application operations.

The string analysis algorithm includes the emulation of string flows through an application. In case of a large enterprise application this process becomes complex, which leads to relatively long execution times (typically the total per-application processing takes 10–100 times longer than one stop function times listed in Table 5). Fortunately, we perform this analysis off-line after we have fetched the application code and related configuration files. Therefore, the amount of time required to perform the analysis is tolerable. In addition, the off-line nature of analysis allows us to use powerful servers. In particular, for our large-scale experiments with all the applications, we used a set of IBM System p5 575 16 core systems that are a lot more powerful than the server used for Table 5. We used a commodity server in this section of the paper just to make it easier for all readers to “feel” the execution times.

Application	Classes	Nodes	Rules	Time (sec)
App1	71	822	11130	33
App2	835	7828	545792	150
App3	1585	11648	406645	173
App4	3141	20510	674450	319

Table 5: Performance on applications of various sizes

7. CONCLUSIONS

We have presented a static discovery method for analyzing code dependencies on external resources such as databases, messaging queues, files, and whatever is not part of the program code. The method is sound under reasonable assumptions. We implemented it for Java EE applications in IBM WebSphere and Oracle WebLogic application servers and analyzed three enterprise application environments, comprising 1097 unique applications. The resulting statistics show that such code analysis is indeed necessary, although the Java EE standards recommend that dependencies should be defined in special resource files and not in the code. The percentage of hard-coded dependencies in the three environments, even if one only counts databases and messaging queues, range from 31 to 94%, i.e., for this percentage of applications all prior static discovery tools, including our own Galapagos tool, miss dependencies. Hence, every transformation project or problem resolution on such environments must take hard-coded dependencies into account. We also showed that less than 10% of hard-coded dependencies are given as string constants, making static string analysis a necessary ingredient. Furthermore, we saw that components of these strings often come from other resources in the environment of the program. Thus, our framework comprises the novel aspect of configuration analysis, which we implemented integrated with the existing string analysis. Configuration analysis may be relevant also outside dependency analysis, e.g., when string analysis is used for security purposes as in [11].

Acknowledgments.
The authors would like to thank Norbert G. Vogl, Daniel A. Prener, Murthy V. Devarakonda, HariGovind V. Ramasamy, and Igor Peshansky for their invaluable contributions to this paper.

8. REFERENCES

[1] HP discovery and dependency mapping (DDM) inventory software. `www.hp.com/hpinfo/newsroom/press_kits/2007/softwareuniversebarcelona/ds_inventory.pdf`.

[2] ISI-snapshot... agent-less accurate and rapid IT infrastructure inventory, configuration and utilization collection using a single tool. `www.isiisi.com`.

[3] Alex Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35:79–111, 1999.

[4] M. Bowker, B. Garrett, and B. Laliberte. EMC smarts application discovery manager. Technical report, ESG Lab Validation Report, July 2007.

[5] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 15–28, 2004.

[6] Alexandru Caracas, Dimitrios Dechouniotis, Stefan Fussenegger, Dieter Gantenbein, and Andreas Kind. Mining semantic relations using NetFlow. In *3rd IEEE/IFIP Int. Workshop on Business-Driven IT Management (BDIM)*, pages 110–111, 2008.

[7] Aske Simon Christensen, Anders Moller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of 10th International Symposium on Static Analysis (SAS)*, 2003.

[8] Mihai Christodorescu, Nicholas Kidd, and Wen-Han Goh. String analysis for x86 binaries. In *Proceedings of 6th Workshop on Program Analysis for Software Tools and Engineering (PASP)*, 2005.

[9] Fyodor. *NMAP(1)*, 2003. `www.insecure.org/nmap/data/nmap_manpage.html`.

[10] Dieter Gantenbein and Luca Deri. Categorizing computing assets according to communication patterns. In *Tutorial on Asset Inventory and Monitoring in a Networked World, Conf. on Networking*, pages 83–100, 2002.

[11] Emmanuel Geay, Marco Pistoia, Takaaki Tateishi, Barbara G. Ryder, and Julian Dolby. Modular string-sensitive permission analysis with demand-driven precision. In *Proceeeding of 31st International Conference on Software Engineering (ICSE)*, 2009.

[12] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, 2004.

[13] HP discovery and dependency mapping software. `https://h10078.www1.hp.com/cda/hpdc/navigation.do?action=downloadPDF&caid=9607&cp=54_4000_100&zn=bto&filename=4AA1-4991ENW.pdf`.

[14] Nikolai Joukov, Murthy V Devarakonda, Kostas Magoutis, and Norbert Vogl. Built-to-Order Service Engineering for Enterprise IT Discovery. In *Proc. IEEE Intern. Conf. on Services Computing (SCC 2008)*, volume 2, pages 91–98, 2008.

[15] Java string analyzer (JSA). `www.brics.dk/JSA`.

[16] A. Kind, P. Hurley, and J. Massar. A light-weight and scalable network profiling system. *ERCIM News*, January 2005.

[17] Andreas Kind, Dieter Gantenbein, and Hiroaki Etoh. Relationship discovery with NetFlow to enable business-driven IT management. In *1st IEEE/IFIP Int. Workshop on Business-Driven IT Management (BDIM)*, pages 63–70, 2006.

[18] K. Magoutis, M. V. Devarakonda, and K. Muniswamy-Reddy. Galapagos: Automatically Discovering Application-Data Relationships in Networked Systems. In *Proc. 10th IFIP/IEEE Intern. Symp. on Integrated Network Management (IM '07)*, pages 701–704, 2007.

[19] Kostas Magoutis, Murthy Devarakonda, Nikolai Joukov, and Norbert Vogl. Galapagos: Model-driven Discovery of End-to-End Application-Storage Relationships in Distributed Systems. *IBM J. Research and Development*, 52:367–378, 2008.

[20] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of 14th International World Wide Web Conference (WWW)*, 2005.

[21] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252, 1977.

[22] C. R. Rich. TADDM's flexible approach to discovery. Technical report, IBM Corporation, July 2007. `ftp.software.ibm.com/software/tivoli/whitepapers/TADDM_s_Flexible_Approach_to_Discovery.pdf`.

[23] Soot: a Java optimization framework. `www.sable.mcgill.ca/soot`.

[24] Tivoli Application Dependency Discovery Manager. `www.ibm.com/software/tivoli/products/taddm`.

[25] R. Vall, E. Phong, C. Etienne, G. Laurie, H. Patrick, and L. Vijay. Soot - a Java bytecode optimization framework. In *Proceedings of IBM Center for Advanced Studies Conference (CASCON)*, 1999.

[26] Watson libraries for analysis (WALA). `wala.sourceforge.net`.

[27] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the SIGPLAN '07 Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[28] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, 2008.

[29] Xu Zheng, Ming Zhan, Z. Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *Proc. 8th Symp. on Operating Systems Design and Implementation (OSDI 2008)*, pages 117–130, San Diego, CA, December 2008.