Software Analysis in the Semantic Web

Joshua Taylor, Robert T. Hall
{taylorj,hallr}@ainfosec.com
Assured Information Security, 153 Brooks Road, Rome, NY USA

ABSTRACT

Many approaches in software analysis, particularly dynamic malware analysis, benefit greatly from the use of linked data and other Semantic Web technology. In this paper, we describe AIS, Inc.’s Semantic Extractor (SemEx) component from the Malware Analysis and Attribution through Genetic Information (MAAGI) effort, funded under DARPA’s Cyber Genome program. The SemEx generates OWL-based semantic models of high and low level behaviors in malware samples from system call traces generated by AIS’s introspective hypervisor, IntroVirt™. Within MAAGI, these semantic models were used by modules that cluster malware samples by functionality, and construct “genealogical” malware lineages. Herein, we describe the design, implementation, and use of the SemEx, as well as the C2DB, an OWL ontology used for representing software behavior and cyber-environments.

Keywords: Linked Data, Malware Analysis, Hypervisor, Introspection, Dynamic Analysis, Software Analysis, Event Correlation, Reasoning, Semantic Web, OWL

1. INTRODUCTION

In this paper, we describe the Semantic Extractor (SemEx), one component of a suite developed by AIS, Inc. during the Malware Analysis and Attribution through Genetic Information (MAAGI) effort,1 funded under DARPA’s Cyber Genome program. The SemEx component of MAAGI is responsible for extracting semantic descriptions of the invocations in a software trace gathered during execution. Initially, and during much of the work described in this paper, we used Zero Wine, and then the Cuckoo sandbox, though later, and currently, we are using an environment based on AIS’s IntroVirt™.2 The semantic descriptions of software traces, coupled with descriptions of the hosts on which the software was executed, were subsequently used as inputs for behavioral analysis, in which malware behavior was characterized semantically based on the particular high-level behaviors it exhibited, regardless of implementation. We begin with a brief description of the test environments we constructed, first with Cuckoo, and then with IntroVirt. We continue with a discussion of the vocabularies we created, as OWL ontologies, and used for semantically representing software traces. We conclude with a review of related work in dynamic analysis using linked data and ontological representations, and consider future directions for this research.

2. MAAGI

The MAAGI effort aimed to aid malware in a number of ways, one of which to provide a suite of tools for the analysis of malware based on semantic analysis of dynamically generated traces of malware execution. Desired analysis included behavior-based clustering, and genealogical classifications of malware families. A number of analytical modules have been explored and implemented in the course of MAAGI, including differential analysis modules, +Hidden Markov Models for identifying malware evolutionary changes, and systemic functional grammars for understanding the intended function of malware.3 MAAGI has also prompted the development of new probabilistic models of malware phylogeny, and probabilistic classifiers for characterizing attackers.4 Some MAAGI analysis components, such as permutation-based binary signatures, are based on static analysis of malware samples, but most of the MAAGI tools that perform some type of dynamic analysis are driven by trace data generated using AIS’s introspective hypervisor, IntroVirt. The overall MAAGI architecture is shown in Figure 1. In the rest of this section, we describe the MAAGI components that support the SemEx and use its results; in the next, we describe the SemEx itself.

*IntroVirt™ — Copyright 2012 Assured Information Security, Inc. (AIS)
Figure 1. The MAAGI architecture aids malware analysts throughout the analysis process. MAAGI automatically preprocesses malware samples through de-obfuscation, and determines the best way to distribute malware samples for distributed analysis. Modular analysis components encapsulate techniques in static reverse engineering, abstract interpretation, trusted disassembly, and differential analysis. Results from the analysis components are then fused together to provide reports to malware analysts.

2.1 Trace Environments

At the core of any dynamic analysis suite there must be a sandbox environment in which malware samples can be executed and observed. Within MAAGI, we restricted our view to trace information, that is, the sequence of function calls executed by a malware sample, and we further restricted ourselves to examining only system calls, that is, we observed only the system calls executed by malware samples, ignoring calls to other libraries, or to other code contained within the malware itself.

2.1.1 Zero Wine & Cuckoo

To avoid developing a trace environment from scratch, we first used the open source Zero Wine environment, a malware analysis environment based on the Wine implementation of the Windows API that allows Windows applications to run on Unix environments. The Semantic Extractor consumed Zero Wine traces after execution of a sample, and produced semantic graphs describing sample behavior. We encountered several issues with Zero Wine, however. One non-trivial issue is that some malware incorporates Wine-detection methods, and only exercises malicious behavior in non-Wine environments. Wine-detection would have proved a significant issue in the long run, but a more pressing issue for us was the Zero Wine required us to implement hooks for almost all the system calls that we were interested in observing. After abandoning Zero Wine, we used Cuckoo, which provided more coverage, as well as automatic provisioning options, though it was still necessary to implement some hooks for system calls.

While both Zero Wine and Cuckoo, provided much of what we needed in a trace environment, neither gave us a principled way to query the operating system, at the time a system call is invoked, to gather context about the parameters passed into a system call, or the value returned from it. For instance, two system calls might receive the same pointer as string arguments, but without the ability to read the memory addressed by that pointer, it was not easy to determine whether the calls were being passed the same string, or if application code had rewritten the buffer with a different string. This problem is magnified with data types that are not necessarily pointers, but identifiers of operating system resources, such as handles. Information about these resources must be gathered during the execution of the monitored software, as many are transient in nature (e.g., handles to temporary files), and it would be impossible to synthesize this data after the execution. To update the handlers provided by these trace environments would have required significant time and effort. To avoid this cost, we transitioned to an environment providing dynamic introspection, AIS’s introspective hypervisor, IntroVirt.

2.1.2 IntroVirt

AIS’s IntroVirt is an hypervisor that allows, through the use of the associated libwintrovirt library, deep introspective access to guest Windows operating systems. Using IntroVirt and libwintrovirt, we were able to intercept malware samples’
invocations of system calls (kernel procedures) and obtain detailed information about system resources on demand. *IntroVirt* can scale to monitor up to a hundred executing virtual machines on a virtual network, and serves very well as a backbone for a trace generation environment.

### 3. THE SEMANTIC EXTRACTOR

The SemEx, shown in Figure 2, is a relatively straightforward MAAGI component that first consumes trace data and constructs a small RDF model from it. The RDF model is then augmented with static knowledge of the domain and by behavioral rules that infer low level behaviors from API calls in the trace, and infer high level behaviors from low level behaviors and other high level behaviors. In this section we give a brief overview of the C2DB, and continue by describing its representations of values, function invocations, and low and high level behaviors.

#### 3.1 The C2DB Ontology

The representations of invocations and low and high level behaviors described here are part of AIS’s C2DB ontology. The C2DB ontology is an OWL ontology version of the C2 (Command and Control) and SA (Situational Awareness) Data Base schema, which was developed under an earlier effort for representing cyber-environments at all layers, from physical components to logical architectures.\(^5\) The C2DB ontology was constructed from the C2DB database schema during AIS’s Cyber Assignment Framework for Experimentation (CAFÉ) effort.\(^6\) In addition to developing the OWL ontology, we have developed an Apache Jena-based object-oriented API for working with C2DB entities that enables developers to model cyber-environments while treating entities (e.g., workstations, servers, software processes) as Java objects. Using the C2DB API, developers can create models without being exposed to the underlying OWL and RDF representations, but still reap the benefits of OWL and rule-based reasoning. We now describe the C2DB’s representations of: invocations, which are typically system calls; low level behaviors, which provide a thin layer of abstraction over invocations; and high level behaviors, which represent arbitrarily complex patterns of behavior. The C2DB vocabulary has been updated since originally conceived; the current version of the C2DB uses the same general structure, but some details differ from what is presented here. The C2DB ontology is an OWL ontology, but in the following sections, we use standard description logic notation to describe C2DB concepts and relations.

#### 3.2 Values & IntentionalValues

For modeling the values that are passed to and returned from system calls, the C2DB ontology contains a number of classes corresponding to Windows API data types, and a number of “intentional values,” which are not API types, but rather the higher level types that users and developers associate with resources. The class Value represents corresponds to API data types, and has subclasses corresponding to primitive machine-word based data types, such as BOOL, BYTE, DWORD, and ACCESS MASK. There is also a subclass CellValue, whose subclasses are pointer types, such as HANDLE, LPDWORD, LPSOCKET, and PLONG. Intentional values, on the other hand, are instances of the IntentionalValue class. IntentionalValue has subclasses such as File, ExecutionThread, FileListing, and RegistryKey. An object property, realizes, associates Values with the IntentionalValues that they represent.

\[
\text{Value} \sqsubseteq \forall \text{realizes}.\text{IntentionalValue}
\]
CellValues, which represent pointers to Values, also contain other Values. The type of Values that can be contained in a CellValue are restricted based on the subclass of CellValue. For instance, LPDWORDs contain only DWORDs, and LPSOCKETs contain only SOCKETs.

\[
\text{LPDWORD} \sqsubseteq \text{CellValue} \land \forall \text{containsValue}.\text{DWORD} \\
\text{LPSOCKET} \sqsubseteq \text{CellValue} \land \forall \text{containsValue}.\text{SOCKET}
\]

### 3.3 Invocations & Functions

Instances of the Invocation class represent the function calls that the trace environment records. In the present work, all Invocations are kernel procedure calls, though any function call can be represented as an Invocation instance. For example, the initial work with Zero Wine and Cuckoo represented user API calls as invocations. Each invocation invokes exactly one function, may realize some (low level) behavior, and returns at most one value.

\[
\text{Invocation} \sqsubseteq = 1 \text{ invokes}.\text{Function} \land \forall \text{realizes}.\text{Behavior} \land \leq 1 \text{ returns}.\text{Value}
\]

The RDF Vocabulary Description Language (RDF Schema, or RDFS) provides vocabulary terms for representing lists and other types of containers. Were the C2DB a simple RDFS vocabulary, it would be natural to associate each invocation with an rdf:list of arguments. However, the RDFS containers are used by the RDF/XML serialization of OWL and cannot be used as containers in OWL ontologies. As a (perhaps inelegant) workaround, this early version of the C2DB contains a hasArgument property with subproperties 1st, 2nd, and so on, up to 15th. This permitted the definition of subclasses of function such as NullaryFunction, 1aryFunction, . . . , up to 15aryFunction. Each of these has a similar definition. For instance, 3aryFunction is described by:

\[
\text{3aryFunction} \sqsubseteq \text{Function} \land = 1 \text{ 1st}.\text{Value} \land \cdots \land = 13.\text{Value} \land = 04.\text{Value} \land \cdots \land = 015.\text{Value}
\]

The Windows API uses annotations in Microsoft’s source-code annotation language (SAL) to mark various properties of function arguments, such as whether they are input arguments, output arguments, input/output arguments, o required arguments. The SAL defines a large number of these annotations, and the C2DB includes several subproperties of hasArgument that help to maintain these annotations: hasInput, hasOutputArgument, isOptional, and isRequired. (The property hasOutputArgument is not named hasOutput, but is a subproperty of hasOutput, which indicates any output of an invocation, including its return value. There are two subproperties of hasOutput: hasOutputArgument, and returns.) The various subproperties of hasArgument are used to constrain and describe the values that are used as invocations to functions.

For instance, corresponding to the function FindFirstFile which has the prototype

```c
HANDLE WINAPI FindFirstFile(
    _In_  LPCTSTR lpFileName, 
    _Out_ LPWIN32_FIND_DATA lpFindFileData
);
```

there is an individual FindFirstFile that has the following type:

\[
2aryFunction \land \forall \text{invokes}^{-1}. (\forall 1st.LPCTSTR \land \forall 2nd.LPWIN32\_FIND\_DATA \land \forall \text{returns}.\text{HANDLE})
\]

The preceding class expression has as instances every binary function each invocation of which must have an LPCTSTR as a first argument, an LPWIN32\_FIND\_DATA as a second argument, and which must return a HANDLE. OWL lacks a way to specify that the individuals related to a FindFirstFile invocation by the 1st and 2nd properties arguments must also be related to that invocation by hasInput and hasOutputArgument, but these relations can be asserted by rules stored with the ontology (e.g., SWRL rules). For FindFirstFile, the desired SWRL rule is

\[
\text{Invocation}(i) \land 1st(i, x) \land 2nd(i, y) \land \text{invokes}(i, \text{FindFirstFile}) \rightarrow \text{hasInput}(i, x) \land \text{hasOutputArgument}(i, y)
\]

While effective, this representation of functions and invocations is somewhat awkward. It was important to maintain the type information of function arguments and to maintain the distinction between direct Values and CellValues, especially when using Zero Wine and Cuckoo, since those trace environments do not provide convenient introspection facilities. In the more recent work with IntroVirt, which provides dynamic introspection, more of the higher level semantic information can be obtained directly. For instance, using IntroVirt, it is possible to examine operating system memory when a system call is intercepted and to determine the type of resource to which a HANDLE refers.
3.4 Low Level Behaviors

The first level of abstraction over Invocations, which are simply OWL encodings of trace data, is provided by low level behaviors. The Behavior class has two subclasses, LowLevelBehavior and HighLevelBehavior. LowLevelBehavior has two disjoint subclasses, FailedBehavior and SuccessfulBehavior, which correspond to failing and succeeding system calls, respectively. While invocations operate on machine-level values, each LowLevelBehavior acts on some intentional value.

$$\text{Behavior} \equiv \text{LowLevelBehavior} \sqcup \text{HighLevelBehavior}$$

$$\bot \equiv \text{LowLevelBehavior} \cap \text{HighLevelBehavior}$$

$$\text{LowLevelBehavior} \subseteq \exists \text{actsOn.IntentionalValue}$$

$$\text{LowLevelBehavior} \equiv \text{FailedBehavior} \sqcup \text{SuccessfulBehavior}$$

$$\bot \equiv \text{FailedBehavior} \cap \text{SuccessfulBehavior}$$

The actsOn property has a number of subproperties, and it is typically one of these subproperties that directly associates a low level behavior with the intentional value that it acts upon. The immediate subproperties of actsOn are closes, creates, deletes, opens, queries, reads, and writes. There are two subproperties of reads: readsCompletely, and readsPartially, which indicate that a behavior read the entire content of a resource, or only part of the content of a resource.

While the existence of some low level behaviors could be inferred from certain OWL axioms, the general task of determining whether an invocation succeeded or failed requires examination of an invocation’s outputs (that is, its output arguments and its return value), and the logic required is often outside the scope of OWL. For each system call that we monitored, we implemented a rule that recognized invocations of the system call and created a corresponding low level behavior (either a failed behavior or a successful behavior) related, by some subproperty of actsOn, to one of the intentional values realized by one of the invocation’s arguments or its return value.

3.5 High Level Behaviors

High level behaviors represent specific behaviors of interest that appear in software traces. Subclasses of HighLevelBehavior include ReadUntilEnd, RegistryKeyWalk, RegistryValueWalk, MakeOutboundConnection, and WalkSystemDirectory. Figure 4 shows, in a general fashion, how a CopyToUSBDrive high level behavior might be inferred from the presence of
Figure 4. A High Level Behavior denoting a malware sample’s self replication is inferred from two other High Level Behaviors. The first hlb-5 completely reads _:b which is the malware sample file, and the second writes that content to a file _:a, a file on a USB drive. The inferred High Level Behavior is a CopySelfToUSBDrive behavior associated with the file _:a.

behaviors that read the entire malware sample and write the content to a file on a USB drive. Figure 5 continues the example from Figure 3 to show how a DirectoryWalk is actually associated with the invocations that realize the invocations that perform the directory walk.

4. SOFTWARE ANALYSIS WITH LINKED DATA

The MAAGI effort itself uses very little publicly available linked data, but the SemEx component has been designed so as to support later incorporation with publicly available data and interoperability with other software processing systems. For instance, given a software sample, more information about the sample could be retrieved from other sources and used to narrow the focus of analysis, for instance, by linking malware descriptions to related resources such as CVE identifiers. MAAGI focused on the analysis of malware under Windows environments, but will eventually support other operating systems and higher, network-wide, levels of behavior, and the use of open-world formalisms such as OWL ensures that the MAAGI is easily extensible, and that large portions of existing work can be applied in new domains with minimal effort. For instance, a Windows invocation is distinct from a Linux invocation, but both are invocations and, once suitable domain knowledge is encoded, both a Linux read system call and a Windows ZwReadFile system call are invocations that realize some low level behavior that reads some resource. Rules that depend only on higher level abstractions, such as “reading from a system configuration file,” will be applicable, regardless of the lower level system calls.

5. RELATED WORK

We have presented the C2DB, an ontology for representing cyber-environments, and described, in part, its vocabulary for representing dynamic traces at varying levels of abstraction. The use of the RDF and OWL in describing cyber-environments is currently popular due to the flexibility of these representation. Nodine et al. designed a Computational Asset Description vocabulary to aid in automatically constructing test environment for experimentation in the National Cyber Range (NCR). Their hardware and software configuration domain knowledge is extensive. It does not, however, include vocabulary for the description of dynamic traces. Al Haider et al. have developed an OWL ontology for representing dynamic traces that encodes traces semantically, using classes such as ConstructorEntryEvent, MethodExitEvent, and ExceptionEvent. This representation, while thorough, seems aimed toward handling complete stack traces, whereas in the present work we have focused on intercepting only operating system calls. Their dynamic analysis work focused on statistical types of metrics, though once a trace has been encoded using their ontology, any number of analytic techniques could be applied. Hoffman et al. describe similar analytic techniques based on presenting abstracted views of traces based on semantic considerations of the events within the trace. They present a formal approach to defining these views, but it is not based on a linked data
representation, so the potential for interoperability with other systems is limited. An important aspect of this research is the use of domain knowledge to guide the higher level views of software traces. In this work, we have described high level behaviors in malware. Malware samples are often small and perform only a few distinct high level behaviors. Belmonte and Dugerdl have examined the more general application of using ontological encodings of domain expertise to assist in software trace analysis. Their approach helps to correlate higher level business process and operational procedures with software executions. There is another type of software modeling that has adopted OWL ontologies for knowledge representation, and that is the modeling of software evolution. Tappolet et al. describe the use of an ontology of software evolution to track the releases and development activity of software projects. This is not a type of dynamic analysis, but we see this orthogonal research as potentially quite useful to dynamic analysis. Combining these approaches, it may be possible to automatically identify interesting attributes of software, for instance, the specific version or release of an application that first exhibited some known vulnerability.

6. CONCLUSIONS AND FUTURE WORK

We have presented an overview of the MAAGI effort, with particular emphasis on the C2DB ontology and the SemEx, and have shown how, using OWL and rules-based reasoning, high level behavioral views of software can be generated from dynamic traces. The work described in this paper is ongoing, and we are committed to further development of the C2DB ontology and the dynamic trace environments that support the analysis exposted here. Indeed, the MAAGI program is a continuing effort, and much of this technology has been advanced an AIS framework for network wide semantic event correlation. We recognize the oppportunities that publically available linked data presents, and we will explore ways to integrate such data sets with MAAGI analytical tools.

ACKNOWLEDGMENTS

This work was supported by a subcontract from Charles River Analytics under US Air Force contract FA8750-10-C-0171, with thanks to Dr. Michael VanPutte and Mr. Timothy Fraser. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.
REFERENCES


