Dynamic Anomaly Detection for More Trustworthy Outsourced Computation

Sami Alsouri, Jan Sinschek, Andreas Sewe, Eric Bodden, Mira Mezini, and Stefan Katzenbeisser

Technische Universität Darmstadt
Center for Advanced Security Research Darmstadt - CASED
Mornewegstraße 32, 64293 Darmstadt, Germany
{sami.alsouri, jan.sinschek, andreas.sewe, eric.bodden}@cased.de,
mezini@st.informatik.tu-darmstadt.de,
katzenbeisser@seceng.informatik.tu-darmstadt.de

Abstract. A hybrid cloud combines a trusted private cloud with a public cloud owned by an untrusted cloud provider. This is problematic: When a hybrid cloud shifts computation from its private to its public part, it must trust the public part to execute the computation as intended. We show how public-cloud providers can use dynamic anomaly detection to increase their clients' trust in outsourced computations. The client first defines the computation's reference behavior by running an automated dynamic analysis in the private cloud. The cloud provider then generates an application profile when executing the outsourced computation for its client, persisted in tamper-proof storage. When in doubt, the client checks the profile against the recorded reference behavior. False positives are identified by re-executing the dubious computation in the trusted private cloud, and are used to re-fine the description of the reference behavior. The approach is fully automated. Using 3,000 harmless and 118 malicious inputs to different Java applications, we show that our approach is effective. In particular, different characterizations of behavior can yield anything from low numbers of false positives to low numbers of false negatives, effectively trading trustworthiness for computation cost in the private cloud.

Keywords: Cloud security, dependability, dynamic analysis, anomaly detection, hybrid clouds

1 Introduction

Cloud computing allows companies to outsource part of their computations to server farms, usually owned by a cloud provider. It promises many benefits, such as reducing infrastructure investments, the ability to quickly adapt its compute power according to the demands (the so-called "elastic cloud"), or the adoption of a pay-as-you-go billing model [22].

But cloud computing comes at a risk. While a company controls its private computer servers, it has limited control over resources rented in the cloud. This motivates the so-called "hybrid cloud" scenario, in which a company owns a private cloud of trusted compute servers, while at the same time this private cloud shares data with a public cloud service owned by a public cloud provider, executing another set of computations. Which computations are performed in the private and which ones in the public cloud depends on the company's preferences and policies.

All instances of hybrid clouds share the common problem that, when shifting computation from their private to their public parts, they must trust the public part to execute the computation as intended. But what justifies that trust?

In this work, we introduce *behavior compliance control*, in which a cloud provider uses methods from dynamic anomaly detection to provide clients trustworthy evidence about the absence of "abnormal" executions caused by incorrect server configurations, version mismatches, hardware glitches or malicious attacks by third parties [20]. Providing such evidence is very important in scenarios where faults or attacks occur through invalid program inputs such as incorrect or compromised configuration files.

Our approach starts with a learning phase in which the client uses an automated tool to learn the behavior of an application by running it in the trusted private cloud on a collection of representative inputs. This process, conducted before outsourcing the application, results in a so-called application model. The model is considered to characterize the application's intended behavior. In this work we study and compare models at different levels of granularity.

After the application has been outsourced into the public cloud, the outsourced application uses runtime monitoring techniques to log critical runtime information into a securely sealed storage, thus yielding trusted evidence on the application's remote behavior. Next, the client verifies if the observed log information, according to this evidence, complies with the application model learned in the learning phase. If the run is found to be compliant, the outsourced computation is assumed to have executed correctly. If the run is not compliant this can be either due to an actual anomaly in the public cloud, or due to a false positive caused by an imprecise application model. We restrict ourselves to deterministic programs, which the client can re-execute in the private cloud to tell both cases apart. If this trusted re-execution yields the same result then the client has identified a false positive, and can use this false positive to refine the application model. If not, then the client has found an anomaly, i.e., an actual piece of evidence of a faulty or maliciously influenced computation in the public cloud.

In this work, we present the first work leveraging dynamic anomaly detection for the scenario of hybrid cloud computing. In particular, we present the following contributions. We present an abstract architectural framework for behavior compliance control. The framework is defined in terms of its abstract security requirements, and hence independent of any concrete implementation. In addi-

tion, however, we present and make publicly available[1] a concrete instantiation of this framework for the Java platform. In this instantiation, we implement a sealed storage using Trusted Computing technologies.

Another main contribution is an empirical evaluation showing how the efficacy of our approach depends on the choice of application model. We evaluate three kinds of models that abstract from an application's dynamic behavior with increasing granularity, by recording (1) the set of called methods, (2) a dynamic call graph, or (3) a dynamic calling context tree. We used our Java-based implementation to produce and evaluate application models for three different open-source applications, applied to 3,000 publicly available documents we believe to be harmless and 118 known malicious documents containing web exploits. Our results show that our approach is effective. In particular, different choices of models can yield anything from low numbers of false positives to low numbers of false negatives. This gives clients a large degree of freedom in trading increased trustworthiness for increased computation cost in the private cloud.

The remainder of this paper is structured as follows. In Section 2, we describe our three choices of behavioral abstractions. Section 3 defines our architectural framework for behavior compliance control, while Section 4 describes our concrete instantiation for Java. In Section 5, we discuss our empirical evaluation, assessing the usefulness of our three abstractions for the purpose of behavior compliance control, as well as the performance of our approach. We discuss related work in Section 6 and our conclusions in Section 7.

2 Characterizing Behavior

Behavior compliance control builds on techniques from dynamic anomaly detection [8,9,11,13,18], a methodology that attempts to detect anomalous executions by comparing certain execution characteristics with those known to be characteristic for correct and "compliant" executions. Our technique significantly extends traditional anomaly detection by a means to conduct the detection process in a distributed but nevertheless trustworthy fashion. Yet, an important design decision that both previous approaches as well as ours have to make is how to best characterize an application's runtime behavior.

Since all previous approaches describe behavior at one level of granularity and have therefore some disadvantages, we decided to not restrict ourselves to a single mind set: Instead of fixing one given classification of behavior upfront, we decided to implement three white-box abstractions on different levels of abstraction, and to compare their relative usefulness for the behavior compliance control of outsourced applications. Clients can then choose which abstraction best fits their needs.

[1] Our implementation is available, in source, along with all our raw data and scripts to reproduce our empirical results, at `http://seceng.de/research/projects/bcc`

Behavior Models. We regard function calls as a main ingredient for characterizing behavior.[2] Consequently, we have evaluated three approximations of behavior by tracing which functions a program calls during its execution, and in which contexts. Each approximation thereby induces a different kind of application model for our behavior compliance control approach. We distinguish models according to the amount of information that they contain (from least to most):

- **Functions:** A set of functions F the application called during the execution.
- **Call graph:** A call graph, with nodes representing functions, and an edge from f to f' if f calls f' at least once during the execution.
- **Calling context tree:** A calling context tree [1], with the root node representing the program's entry point and a node f' as child of node f if f calls f' in the same context at least once during its execution.

To illustrate these abstractions, consider the example program in Figure 1. Figure 2a shows the "Functions" representation of the example program. Herein, the model just consists of the set of all functions called during the program's execution. Figure 2b, on the other hand, shows the program's dynamic call graph. Note that in a call graph, every function, such as `bar`, is represented by exactly one node, no matter in how many different contexts the function is invoked. Figure 2c shows the program's calling context tree. In this representation, calling contexts are kept separate: Because `bar` is called by two different functions, once by `main` and once by `foo`, it appears twice in the tree, just under the appropriate contexts.

We chose these three different characterizations of behavior carefully, so that one can construct a model of higher abstraction from a model of lower abstraction. This allows us to compare the models directly to each other, based on the very same data set. The fact that the three different abstractions form such a total order allows us to evaluate different characterizations of behavior at opposite ends of the granularity spectrum: The "Functions" abstraction is quite coarse-grained but can be computed very efficiently. Yet, by its nature it may have the tendency to yield false negatives, i.e., to miss anomalies. Hence, the amount of trustworthiness that this abstraction provides is relatively low. The calling context trees at the other end of the spectrum are very fine-grained. Their computation consumes more time, and by their nature they tend to cause a relatively large number of false positives, increasing the necessary computation cost in the private cloud. But nevertheless, this may still be a price worth paying for the additional trustworthiness they provide. In Section 5, we present an extensive evaluation demonstrating the absolute and relative utility of those abstractions for the task of behavior compliance control. We formalize our abstractions in the appendix.

[2] We use the term "function" instead of "method" because our approach is not bound to Java. Our functions are not "functional" in the strict sense: They may have side-effects.

```
1 public static void main(String args[]) {
2     foo();
3     bar();
4 }
5
6 static void foo() { bar(); }
7
8 static void bar() { }
```

Fig. 1: Example program

{main, foo, bar}

(a) Functions

(b) Call graph (c) Calling context tree

Fig. 2: Three abstractions of the example program

Model Generation in the Learning Phase. For the purpose of behavior compliance control, models should be sensitive to malicious inputs, where faults or attacks occur through them, such as incorrect or compromised configuration files.

We therefore opt for a dynamic approach that collects an application model as a union of a set of runtime execution profiles. Within the trusted private cloud, the client collects an execution profile for every test run. The application's final model for this training data is then defined as the union of all those individual profiles. In the case of Functions we use simple set union, while in the case of call graphs or calling context trees we define the union in the natural way, by computing the union over the graph's, respectively tree's, node and edge sets. We call the resulting profile the application's *model*. Since the union operation is associative, one can compute the model in a step-wise and iterative way, i.e., after each individual profile is collected, or instead compute the union once over all collected individual profiles.

This property is key to our approach: When a model appears too restrictive, it can easily be expanded by joining the application's current model with new execution profiles. Clients can make use of this property after having identified a false positive. The model is extended accordingly, to avoid the same false positive in the future. This process can be fully automated.

3 Platform Architecture

In this section, we present our abstract platform architecture for behavior compliance control. The architecture assumes the presence of a trusted logger that is sufficiently tamper-resistant, as well as securely sealed storage on the host that performs the computation. Assuring the integrity of these components is a problem complementary to the one of behavior compliance control and may be achieved through several means. In Section 4 we will describe a concrete instantiation of the generic architecture that fulfills these requirements, including concrete mechanisms for establishing the integrity of collected profiles.

As described in Section 2, the client first computes an application model by running the software in a trusted environment, the private cloud. Subsequently, the client outsources the application and executes it in the untrusted public cloud. After execution, the public cloud provides the client with evidence about the application's behavior. The client finally verifies the evidence locally to decide on its trustworthiness, and, if required, refines the application model. In the following, we detail the individual phases of this procedure.

I: Learning phase. As described in Section 2, the client generates an application model m, characterizing the behavior of the application, by running the application in the client's trusted private cloud collecting the generated profiles. Afterwards, the application is outsourced to the public part of the cloud.

II: Runtime phase. Figure 3 shows the abstract platform architecture of the hosting platform. We assume the presence of trusted system measurement components, which assure the load-time integrity of the loaded applications and the trusted logger. The load-time integrity of the public cloud platform can be verified by the client before outsourcing takes place. Those mechanisms in fact assure the client that, at load time, exactly those components are brought to execution that the client intended to execute.

The trusted logger logs the events coming from the application itself (i.e., in case of using instrumented code) or from the middleware (e.g., the Java Virtual Machine, or a business process engine), on which this application runs. As mentioned previously, a core task of the public cloud is to generate logs about the execution of the outsourced application in a trustworthy way. As the generated logs are security critical, secure storage is required.

III: Compliance verification phase. Once the client has obtained the log and verified its integrity, he compares the log against the application model collected in the learning phase. We write $l \models m$ if the log l corresponds to the model m and $l \not\models m$ otherwise. Whenever $l \not\models m$, this means that the log diverged from the model, indicating a dubious execution. Such a divergence could be the effect of an execution anomaly but could also just be a false positive, due to an overfitting application model.

To tell apart a false positive from an actual anomaly, the client would then re-execute the application in the private cloud and record the resulting log l'. We

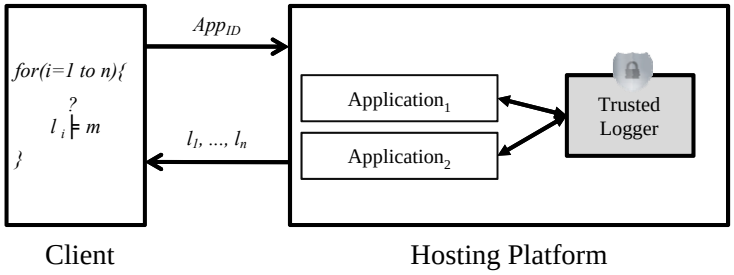

Fig. 3: Our architecture for behavior compliance control

write $l \equiv l'$ if the log l is equivalent to the log l' and $l \not\equiv l'$ otherwise. Whenever $l \equiv l'$, the execution in the public cloud is considered correct and trustworthy, and the divergence was a false positive. In this case, the client expands m by including l. By doing so, m can be continuously improved to decrease the overall false positives rate. (Note that there is a trade-off: By including l in m the model also becomes more permissive, which may yield more false negatives as well.) If otherwise $l \not\equiv l'$, the execution is considered untrustworthy. Concrete implementations of the "$\models$" and "$\equiv$" operators depend on the kind of profile being used. We discuss our concrete instantiation in Section 4.

Note that, in contrast to intrusion detection, where a very low false positive rate is imperative, our approach can tolerate higher rates. Essentially, the false-positive rate determines how much outsourced computation has to be re-done in the private cloud. Thus, in the hybrid cloud environment, a false-positive rate in the order of a few percent may well be acceptable, as still the bulk of the computation is performed in the public cloud.

Threat model and limitations. Our approach hinders attacks by the cloud provider but cannot fully prevent them. This is because we demand the existence of a trusted logger, whose integrity can only be assured through specialized hardware. The cloud provider has access to this hardware and thus may have the means to compromise it. Assuming, however, that the trusted logger can indeed be trusted, our architecture guarantees that detected anomalies can be communicated to the client in a tamper-proof way. Our approach can effectively identify execution anomalies caused by malicious program inputs of any kind, of by faults or misconfigurations in the execution environment. The former is particularly useful to identify attacks on such systems in the public cloud that have a public interface, for instance web servers or document servers.

Our approach is passive, i.e., anomalous behavior is detected only after the fact. As such, our approach cannot prevent anomalous behavior from happening. Instead it allows the client to identify the anomaly, and thereby to re-execute the original computation (and thus circumvent the anomaly) in the trusted private cloud, and to take other appropriate measures such as legal actions in case the anomaly was caused by a malicious intruder. While approaches to active compliance control are possible (e.g., by inserting a runtime monitor that checks

model compliance just in time), such an approach would greatly suffer from any false positives: When a monitor detects an anomaly at runtime, it must decide whether the anomaly is real or a false positive at that point in time. In most cases, this is impossible. In any case, such active compliance control would not be able to provide the flexible trust/cost trade-off that we see as one of the greatest benefits of our approach.

Last but not least, it should be noted that all security-related approaches to anomaly detection, including our own one, are susceptible to mimicry attacks [30, 36], in which an attacker tries to execute behavior that is malicious, but nevertheless mimics legal behavior in such a way that the malicious behavior remains undetected. This problem can be mitigated somewhat by keeping the application model undisclosed, but to the best of our knowledge no way to absolutely avert mimicry attacks is known to date.

4 Platform Instantiation

We next discuss our instantiations of the generic architecture described in Section 3 to the Java language and platform. As a way to provide secure storage, we base our instantiation on concepts from Trusted Computing. Our full implementation of this instantiation is available online, in source, on our project website.

4.1 Adaption to Java

To generate execution profiles, we use JP2, an open source calling-context-tree profiler for Java [26, 27]. This light-weight profiler consists of a small Java agent, which instruments the profiled application at load time, and an accompanying tool to instrument the Java runtime library ahead-of-time. This combination enables us to generate execution profiles which cover not only the application but also the Java runtime library itself. Moreover, JP2's profiles cover not only methods that have a bytecode representation but also method calls made in either direction across the bytecode-native code boundary. The following details are specific to a Java-based setting:

- **Virtual machine-based execution:** The Java platform allows for easy load-time transformation of code. Hence, to introduce a runtime monitor, a client does not need to instrument his application in house. Instead, the application can be transformed remotely, by a custom class loader [23] or transformation agent. Such instrumentation is performed on the level of bytecode and requires no access to source code. JP2 does exactly this.
- **Generated code:** The same class-loader mechanism that makes it easy to introduce a runtime monitor at load time also makes it possible to generate classes at runtime. Such classes frequently bear a randomized name, and that name must be canonicalized to ensure that the same method, up to renaming, can be reliably identified across program runs. To that end, we

integrated the hashing facility from TamiFlex [5] with the calling-context-tree profiler described next.

- **Recursion:** When using the Calling Context Tree abstraction, recursive calls can cause the profile to grow very large. One way to address this would be to "fold" those sub-trees in the CCT that exhibit a recursive structure. The generated profiles would hereby be bounded. However, what exactly counts as recursion in a language with dynamic dispatch is not obvious: Do only calls to the same target method count or also calls to a different target method of the same call site? Calls of the latter kind are frequent, e.g., when operating on objects structured using the Composite pattern [10]. Moreover, mutual recursion or, more generally, larger cycles of calls could be considered recursive as well and maybe thus subject to folding. For the purpose of this paper we restrict the discussion to the straight-forward calling context tree abstraction produced by JP2 and do not fold recursive calls; thus, the tree structure mirrors the entire computation.

In our current implementation, we always collect full calling context trees (CCTs), even if we are just interested in call graphs or function sets. Call graphs are computed from a CCT by merging nodes with the same name, and method sets are computed by a simple exhaustive search through the call graph. This methodology is a limitation of our prototype. For efficiency, a realistic implementation would record only the information required for the chosen behavior characterization. We implement the "$\models$" operator from Section 3 by simply checking whether the calling context tree, call graph or function set collected on the server is a sub-tree, sub-graph or sub-set of the respective application model. For the "$\equiv$" we define that $l \equiv l'$ if the respective trees or graphs are isomorphic, or in the case of function sets if they are equal. We store calling context trees and call graphs in a normalized fashion that allows us to decide $l \equiv l'$ in time linear in the size of the operands.

4.2 Integrity of Trusted Components & Runtime-Secure Storage

To safeguard not only against anomalies caused by accidental misconfigurations or hardware glitches but also against (certain classes of) malicious attacks, it is necessary to store the runtime information collected in a trustworthy manner.

Our particular choice to instantiate the integrity measurement components and the secure storage relies on the concepts of Trusted Computing. To assure load-time integrity of the trusted logger in the public cloud, we first build a chain of trust, starting from the cloud server's hardware up to the trusted logger itself. For this purpose, our hardware was equipped with a TPM chip. Trusted boot is assured using the Grand Unified Bootloader (GRUB) version 0.97 together with TrustedGrub [35] version 1.1.5. We used the attestation framework IMA (Integrity Measurement Architecture [25]) to allow the client to verify the load-time integrity of the behavior measurement component and the secure storage (effectively comparing cryptographic hashes of the binaries). Clients would

typically use the integrity reporting facilities of those components before the outsourcing of computations takes place.

To provide a runtime-secure storage, our logging facilities record a hash chain of all logged events in one fixed Platform Configuration Register (PCR) of the TPM chip. For each logged event, the register's current hash value is replaced by a hash over this current value and the event's own payload data. A client can then validate the integrity of the log by re-performing the same hash operation on the log and comparing the resulting hash values. If they differ, the log has been tampered with, and the computation should be re-performed in the private cloud.

In the general context of outsourced applications, the use of a single hardware TPM is insufficient: Many applications execute in the same remote host, and each can be executed many times. Data measured for different application must be stored separately. We hence use the concept of virtual TPMs (vTPMs), which allows us to assign a (unique) virtual TPM instance to each outsourced process [28]. All vTPMs are managed by a vTPM manager, which provides an interface to create and access vTPM instances; the vTPM manager is notified whenever an application instance is started. We implemented a vTPM manager in Java as a proxy to create and manage vTPM instances. The vTPM instances themselves are implemented using the TPM emulator proposed by Strasser and Stamer [29]. To communicate with vTPMs, we use the tpm4java library [33], which facilitates using the cryptographic functionalities of vTPMs in our Java applications. In detail, one chooses a particular vPCR i to hold a hash chain of all recorded events. Whenever a new log entry is generated, the vPCR i is extended by hashing the log entry using SHA-1 and running the `TPM_Extend` command of the corresponding vTPM instance as described in the TPM specification [34]. The log entry itself is stored in external (untrusted) storage. Thus, after the outsourced application terminates, the vPCR register i of the vTPM associated to the application contains a (securely stored) hash chain of all recorded events; further, the log l of all events is available on storage.

Subsequently, remote attestation is performed to securely transfer the log (which is signed by the vTPM) to the client. After verifying the log's integrity, the client verifies the compliance of each single log entry (i.e., each call edge) with the application model as described in Sections 2 and 3.

5 Evaluation

In this section we evaluate our three behavior abstractions from Section 2, function sets, call graphs, and calling-context trees, with respect to following four research questions:

RQ1 (Feasibility): In the learning phase, do the collected profiles converge to a stable model of legal inputs with low false-positive rates?

RQ2 (Effectiveness): To what extent is the application model able to discriminate between legal and illegal inputs?

RQ3 (Scalability): Is the profile size independent of the application's runtime?
RQ4 (Efficiency): Can our approach be implemented efficient enough to induce a sufficiently low runtime overhead?

5.1 General Experimental Setup

One restriction of our approach is that, to produce representative models, it requires a representative set of program inputs. This restricted us in our choice of evaluation subjects; we had to opt for applications for we would be able to obtain large sets of abnormal/malicious as well as legal/harmless inputs. We chose the following subjects:

1. *Apache pdfbox:* A PDF manipulation framework [2].
2. *POI-HSLF:* A Java API to extract data from PowerPoint documents [3].
3. *POI-HWPF:* A Java API to extract data from Word documents [3].

All applications operate on popular file types (Adobe PDF, Microsoft PowerPoint `.ppt`, and Microsoft Word `.doc`), all of which can be obtained in large numbers from the web. Moreover, all three file types are well-known attack vectors. For the PDF file type there further exist repositories of malicious inputs, which serve us to simulate possible manipulations by the cloud provider (details below).

5.2 RQ1: Feasibility

For behavior compliance control to be feasible, it must be possible to automatically generate a useful application model from only a number of representative inputs small enough not to be prohibitive. Moreover, the generated application models must yield false positive rates low enough for the approach to pay off. Remember that any false positive induces increased computation cost in the private cloud.

For our evaluation, we used the top 1,000 results of a Google search for `filetype:pdf`, `filetype:ppt`, and `filetype:doc`, respectively. The resulting corpus of inputs allowed us to generate application models from various numbers of input documents. We believe those 3,000 documents to be harmless, legal documents.[3] Therefore, if a model classified any run as abnormal that was induced by one of those inputs, we count this classification as a false positive.

To generate the application models, we first used the JP2 profiler (cf. Section 4.1) to obtain a calling context tree for each of the applications and inputs. From the resulting CCTs we then derived both dynamic call-graph and function-set profiles. This ensures that, for a given input document, all three abstractions of the application's behavior are consistent.

[3] This is because Google has put in place filters to remove invalid or potentially malicious documents from its search index. In fact we tried to find malicious documents using Google but failed.

We then used ten-fold cross-validation [21] to determine the false-positive rate that can be expected of the collected models. For each of the three file types, the 1,000 profiles were first divided into ten subsets of 100 profiles each. Then, each profile from one of the subsets was checked for compliance with application models derived from an increasing number of (randomly chosen) profiles in the other nine subsets, up to all 900 profiles in the end. Every compliance check yields either the answer "compliant" or an anomaly warning. Since we consider our training set to only contain compliant input documents, we consider all warnings to be false positives.

Figure 4 shows the resulting false positive rates, averaged over the 10 subsets, for various training set sizes. Because we used ten-fold cross-validation, at most 900 out of the 1000 available inputs were used for model generation. As Figure 4 shows, for both the Function and Call Graph abstractions it suffices to use only a few hundred inputs for model generation to obtain a model with a false-positive rates below 5%. Using the calling-context-tree (CCT) abstraction, however, requires a larger number of inputs to achieve low false-positive rates. Even when using 900 inputs to generate the application model, an average of about 22%, 10%, and 3%, respectively, of the remaining 100 profiles are deemed non-compliant. We also observe that at least for the Calling Context Tree abstraction the false-positive rates very much depend on the program under evaluation.

5.3 RQ2: Effectiveness

To increase trustworthiness, behavior compliance control must be able to detect abnormal execution behavior. For the purpose of our evaluation we consider an execution to be abnormal if it executes on an abnormal program input. In reality, there could be other sources of abnormality such as glitches in the hardware or execution environment. We obtained abnormal inputs from two distinct sources: from dedicated repositories of malicious inputs for the file types in question and from applying fuzzing techniques to legal inputs.

To simulate a targeted attack by a third party, we have used a set of 118 PDFs that have previously been used in exploits.[4] For this experiment, we used application models computed by including all 1,000 profiles for PDF file type. As Table 1 shows, all abnormal executions were classified correctly when using the Calling Context Tree abstraction. When using the more coarse-grained Call Graph and Function abstractions, however, only 34 % respectively 11 % of inputs were classified correctly. We therefore conclude that it is essential to use information-rich profiles to detect targeted attacks reliably. This is the main trade-off at the heart of this paper: increased trust requires an increase investment to counter-balance the increased rate of false positives caused by such information-rich profiles.

[4] Test data taken from `http://contagiodump.blogspot.com/2010/08/malicious-documents-archive-for.html` (Collection 3).

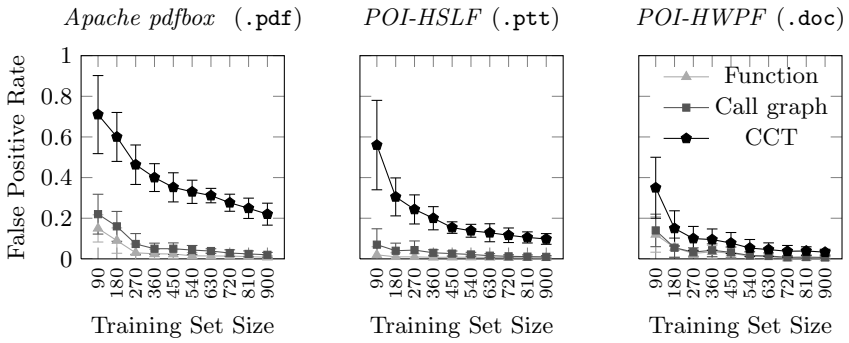

Fig. 4: False positive rate for differently-sized training sets (arithmetic mean ± standard deviation of 10 training sets each).

	Exploits	Fuzzed		
	Apache pdfbox (`.pdf`)		*POI-HSLF* (`.ppt`)	*POI-HWPF* (`.doc`)
Functions	11 %	83 %	100 %	100 %
Call graphs	34 %	89 %	100 %	100 %
CCTs	100 %	97 %	100 %	100 %

Table 1: Percentage of inputs (exploits or fuzzed) detected as illegal.

As we were unable to obtain a similarly large number of malicious PowerPoint and Word documents to simulate a targeted attack, we commenced on a best-effort basis and resorted to fuzzing techniques to simulate an untargeted attack or a problem caused by a faulty data transmission. For each file type, we randomly picked 100 documents from of our corpus of legal documents and applied simple fuzzing techniques to them.[5] This process yields 100 documents each which we define to be abnormal inputs. For each of these inputs we then ran the corresponding application and compared its behavior, abstracted as Functions, Call Graph, or Calling Context Tree, with the application model of legal inputs used before.

Table 1 shows the percentage of fuzzed inputs that were successfully detected as illegal. As these results show, false negatives created by this simple fuzzing algorithm are easy to recognize. It follows that abnormal program runs induced by inputs corrupted in this manner will most likely be detected using behavior compliance control; the abstraction chosen (Functions, Call Graph, Calling Context Tree) has little influence on the detection rate. Those observations hold for the particular fuzzing approach we consider. More targeted fuzzing approaches, taking advantage of the input document's internal structure, may be harder to

[5] 10 random single-byte changes beyond the first 1024 bytes of data; the latter avoids corrupting the main document header, a case that is particularly easy to identify as abnormal.

recognize, but from a security perspective would probably also be less capable of exploiting a vulnerability in the outsourced application.

5.4 RQ3: Scalability

For behavior compliance control to pay off, checking for compliance must be affordable, and must scale to large, long-running applications. We thus evaluate whether the size of the model correlates with the runtime of the application. If this were the case, the compliance check could be as expensive as re-performing the actual outsourced computation, hence defeating the purpose of outsourcing.

For the Function and Call Graph abstractions it is immediately obvious that no such correlation can exist. This is because the number of functions, and consequently the number of call-graph edges, is statically bounded. For the Calling Context Tree abstraction, however, this is not the case. In particular, the use of recursion can cause an application's calling context tree to to any size.[6] Figure 5 visualizes the relation between application runtime (with CCT logging enabled) and and the number of nodes in the resulting CCT profile. Interestingly, in our benchmark longer-running applications do *not* induce significantly larger profiles; thus, our approach scales well over time.

5.5 RQ4: Efficiency

We comment on the runtime overhead caused by the instrumentation necessary for profile generation and on the overhead induced by using securely sealed storage.

For the experiments mentioned above, we used a setup as described in Section 2: we collected calling context trees in all cases, and in a second step computed call graphs and method sets based on the collected trees. This procedure is inefficient. In a real-world setting one would rather opt for a customized instrumentation that emits the respective representation directly, as this can safe a significant amount of execution time. While computing full CCTs will generally incur a significant runtime overhead (10 times or more), one can bring overheads down to under 5% by using probabilistic calling context trees [6]. Such probabilistic CCTs appear quite sufficient for our purposes, and we plan to evaluate their utility in future work. Method sets and call graphs are statically bounded and can therefore be indexed ahead-of-time, which makes instrumentation possible that produces little to no observable runtime overhead [16]. We thereby conclude that sufficiently efficient implementations are possible given the state of the art in dynamic program analysis. While such implementations are outside the scope of this paper, we plan to investigate them in future work.

We measured the runtime cost of our runtime-secure storage on a machine equipped with an AMD Phenom II X2 555 processor and 4 GiB RAM under GNU/Linux (kernel 2.6.32) and the TPM emulator version 0.7.2. Our tests show that the most expensive operation is to create a vTPM instance, which takes

[6] In practice, the virtual machine's maximum stack size does impose a (large) limit.

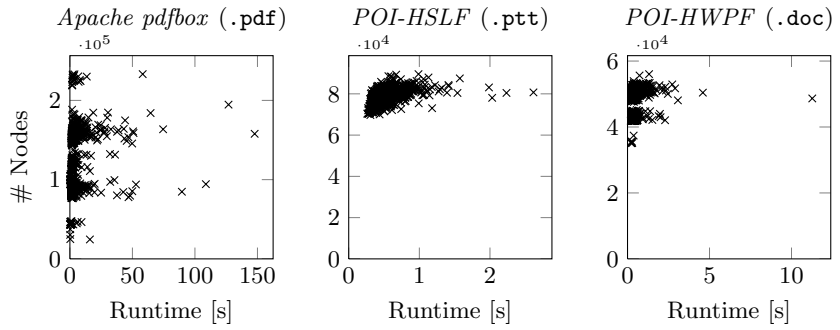

Fig. 5: Relation between application runtime and model size, measured in number of calling context tree nodes.

1 second on average. However, this operation is only invoked once, at application startup time. The overhead is caused by the expensive `TPM_TakeOwnership` operation, which creates the Storage Root Key (SRK) key-pair.

The average total cost of storing a CCT profile depends on the average node number. For pdfbox, POI-HSLF and POI-HWPF those are 120,850, 78,568 and 48,239 respectively. Hashing the unique identifier (8 bytes) of every node takes about 6 μs. The instruction `TPM_Extend`, which extends a PCR register with a hash, takes 400 μs. That is, we estimate the overhead of securely storing a full CCT profile for pdfbox, POI-HSLF and POI-HWPF at about 50, 32 and 20 seconds respectively. When using the more coarse-grained Call Graph abstraction, only an average 5,313, 2,338 resp. 2,289 nodes must be stored for pdfbox, POI-HSLF and POI-HWPF respectively, lasting approximately 3.1, 1.95 and 1.93 seconds. The most efficient abstraction are Functions. The overhead for Functions is 2.1, 1.53 and 1.52 seconds for 2,577, 1,301 and 1,281 functions respectively.

Our results show that the cost of secure storage becomes an issue with CCTs but appears low enough for the other two abstractions. In any case, note that storage can be performed asynchronously on a separate processor core (or even a set of those).

6 Related Work

There is has been a significant amount of previous work on automated property inference [7, 24, 31, 32] and anomaly detection [13, 18] on many different levels, both static and dynamic, all with their relative strengths and weaknesses. Many of those approaches could be integrated into our generic architecture defined in Section 3. We decided to define our own set of three behavior abstractions because this setup would allow us to evaluate the relative properties of those abstractions. Our approach extends all previous approaches to anomaly detection by allowing anomalies to be identified in a distributed but trustworthy manner.

Our approach is not the first to capture program behavior in terms of calling-context information. Ammons et al. [1] show how to generate context-sensitive performance profiles efficiently, using hardware performance counters. Dynamic sandboxing, proposed by Inoue et al. [19], shows similarities with behavior compliance control. Like behavior compliance control, dynamic sandboxing relies on dedicated training runs to determine a set of legal behaviors. However, Inoue et al. only consider profiles at function granularity and validate them in two very limited scenarios; in particular, they do not provide a detailed, quantitative evaluation and do not consider a broader applicability of dynamic sandboxing beyond runtime monitoring.

Our approach builds on ideas from intrusion detection. In the mid-nineties, Forrest et al. [9] addressed an important problem in intrusion detection, the definition of what they call "self", in other words a system's normal behavior. The authors propose a method to define "self" for privileged Unix processes by recording short sequences of system calls. Behaviors that deviate from these patterns are flagged as anomalous and considered untrustworthy.

None of those approaches considers the scenario of behavior compliance control in cloud computing. In addition, all approaches are black-box approaches (in addition to other similar works mentioned in Section 2). Compared to our approach, this gives them the advantage of being independent of any programming language or compiler. On the other hand, white-box approaches such as ours yield higher flexibility (as they can obtain more information) and finer granularity.

Other authors have proposed enforcement architectures to control access to data objects distributed to remote systems [37]. Such architectures control how outsourced applications can access outsourced objects at runtime, assuming that these applications are trusted after verifying their load-time integrity. As we discussed before, behavior compliance control goes well beyond such load-time based measures.

Trusted Computing allows to remotely attest the integrity of computing platforms. Behavior compliance control goes beyond binary attestation by not only considering the integrity of the application's code at load-time, but its actual runtime behavior. Gu et al. [15] propose an approach to remote attestation that can be seen as complementary to ours. Behavior compliance control is focused on assessing the compliance of a single application's execution to its model. Gu et al.'s approach, on the other hand, rather focuses on system-wide attestation; the authors attest behavior by measuring the ways in which different processes call each other. In an approach called Semantic Attestation, Vivek et al. [17] propose to use a trusted virtual machine for remote attestation. The core idea is that such a trusted virtual machine is capable of performing code analysis and runtime monitoring. In the approach, the appropriate property checkers need to be programmed manually, though. This is in stark difference to behavior compliance control, in which application models are automatically generated from legal executions. In more recent work, Gu et al. [14] propose an architecture to attest the execution of single mission-critical subroutines of an outsourced application.

The authors use the debug facilities of certain CPUs to track the execution of a specific function. The execution of the function is then transferred to a secure environment prepared by a secure kernel.

Finally, some effort has been spent on the construction of schemes for verifiable computation [4, 12], which aim at outsourcing computations to a third party, while offering a proof of correctness for the result. At the moment, these constructions are rather impractical and cannot cope with side-effects of the program execution.

7 Conclusion

We have presented behavior compliance control, a novel approach to increase the trust in the validity of executions of outsourced applications. The approach goes beyond load-time based systems for compliance control by considering the application's runtime behavior. This allows the client outsourcing the application to detect abnormal executions even in cases where the application's code remains unaltered after loading. Such anomalies can for instance be caused by faulty or malicious inputs, misconfigurations, version mismatches or hardware glitches. We have presented a reference architecture for behavior compliance control, and an instantiation for the Java platform which is available as open source.

We have implemented and evaluated our approach based on three different abstractions of runtime behavior: function sets, call graphs and calling context trees. Using a large-scale empirical evaluation we could show that the former two abstractions yield few false positives, while still being able to identify a significant number of abnormal executions caused by malicious inputs, and all cases of abnormal executions caused by fuzzed inputs. Those abstractions can also be implemented efficiently. Calling context trees identify all malicious inputs but also yield a larger number of false positives, causing additional computation cost in the client's private cloud. It is hence up to the client to decide whether this additional cost is justified by the increased trust that this abstraction offers.

An interesting piece of future work would be to evaluate optimized implementations of behavior abstractions in the apparent sweet spot between full calling context trees and call graphs. An approach with bounded context strings paired with probabilistic calling contexts [6] appears like a potentially optimal candidate in this solution space.

Acknowledgements. This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE and by the Hessian LOEWE excellence initiative within CASED.

References

1. Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proc. of the 10th Conference on Programming Language Design and Implementation (PLDI)*, pages 85–96, 1997.

2. Apache Software Foundation. The Apache Java PDF Library (PDFbox). `http://pdfbox.apache.org/`.
3. Apache Software Foundation. The Java API for Microsoft Documents (Apache POI). `http://poi.apache.org/`.
4. Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Verifiable delegation of computation over large datasets. In *Advances in Cryptology (CRYPTO 2011)*, volume 6841 of *LNCS*, pages 111–131. Springer Berlin / Heidelberg, 2011.
5. Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proc. of the 33rd International Conference on Software Engineering (ICSE)*, pages 241–250, 2011.
6. Michael D. Bond and Kathryn S. McKinley. Probabilistic calling context. In *Proc. of the 22nd Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 97–112, 2007.
7. Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proc. of the 21st International Conference on Software Engineering (ICSE)*, pages 213–224, 1999.
8. Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *Proc. of the 2003 IEEE Symposium on Security and Privacy (S&P)*, pages 62–75, 2003.
9. S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff. A sense of self for Unix processes. In *Proc. of the 1996 Symposium on Security and Privacy (S&P)*, pages 120–128, 1996.
10. Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
11. Debin Gao, Michael K. Reiter, and Dawn Song. Gray-box extraction of execution graphs for anomaly detection. In *Proc. of the 11th Conference on Computer and Communications Security (CCS)*, pages 318–329, 2004.
12. Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Advances in Cryptology (CRYPTO 2010)*, volume 6223 of *LNCS*, pages 465–482. Springer Berlin / Heidelberg, 2010.
13. Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *Proc. of the 19th International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–130, 2010.
14. L. Gu, Y. Cheng, X. Ding, R. H Deng, Y. Guo, and W. Shao. Remote attestation on function execution. In *Proc. of the 1st International Conference on Trusted Systems (INTRUST)*, page 60–72, 2010.
15. Liang Gu, Xuhua Ding, Robert Huijie Deng, Bing Xie, and Hong Mei. Remote attestation on program execution. In *Proc. of the 3rd Workshop on Scalable Trusted Computing (STC)*, page 11–20, 2008.
16. Tobias Gutzmann and Welf Löwe. Custom-made instrumentation based on static analysis. In *Proc. of the 9th International Workshop on Dynamic Analysis (WODA)*, 2011.
17. Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *Proc. of the 3rd Conference on Virtual Machine Research And Technology Symposium*, pages 3–20, 2004.

18. Sudheendra Hangal and Monica S Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. of the 24th International Conference on Software Engineering (ICSE)*, pages 291–301, 2002.
19. Hajime Inoue and Stephanie Forrest. Anomaly intrusion detection in dynamic execution environments. In *Proc. of the 2002 Workshop on New Security Paradigms (NSPW)*, pages 52–60, 2002.
20. Y. Karabulut, F. Kerschbaum, F. Massacci, P. Robinson, and A. Yautsiukhin. Security and trust in IT business outsourcing: a manifesto. *ENTCS*, 179:47–58, July 2007.
21. Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1137–1143, 1995.
22. Mary C. Lacity, Shaji A. Khan, and Leslie P. Willcocks. A review of the IT outsourcing literature: Insights for practice. *The Journal of Strategic Information Systems*, 18(3):130–146, September 2009.
23. Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *Proc. of the 13th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 36–44, 1998.
24. M. Pradel and T. R Gross. Automatic generation of object usage specifications from large method traces. In *Proc. of the 24th International Conference on Automated Software Engineering (ASE)*, pages 371–382, 2009.
25. Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proc. of the 13th USENIX Security Symposium*, pages 1–16, 2004.
26. Aibek Sarimbekov, Andreas Sewe, Walter Binder, Philippe Moret, and Mira Mezini. JP2: Call-site aware calling context profiling for the Java Virtual Machine. *Science of Computer Programming*, 2012. doi:10.1016/j.scico.2011.11.003.
27. Aibek Sarimbekov, Andreas Sewe, Walter Binder, Philippe Moret, Martin Schöberl, and Mira Mezini. Portable and accurate collection of calling-context-sensitive bytecode metrics for the Java Virtual Machine. In *Proc. of the 9th Conference on the Principles and Practice of Programming in Java (PPPJ)*, pages 11–20, 2011.
28. Vincent Scarlata, Carlos Rozas, Monty Wiseman, David Grawrock, and Claire Vishik. Tpm virtualization: Building a general framework. In Norbert Pohlmann and Helmut Reimer, editors, *Trusted Computing*, pages 43–56. Vieweg+Teubner, 2008.
29. Mario Strasser and Heiko Stamer. A software-based trusted platform module emulator. In *Proc. of the 1st Conference on Trusted Computing and Trust in Information Technologies (Trust)*, pages 33–47, 2008.
30. Kymie M. C. Tan, John McHugh, and Kevin S. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *Revised Papers from the 5th International Workshop on Information Hiding (IH)*, pages 1–17, 2003.
31. S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proc. of the 24th International Conference on Automated Software Engineering (ASE)*, pages 283–294, 2009.
32. S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proc. of the 31st International Conference on Software Engineering (ICSE)*, pages 496–506, 2009.
33. The tpm4java library. `http://sourceforge.net/projects/tpm4java/`.
34. Trusted Computing Group, Inc. *TPM Main Specification Level 2 Version 1.2*, March 2011. Revision 116.

35. The TrustedGRUB extension to the GRUB bootloader. `http://sourceforge.net/projects/trustedgrub/`.
36. David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proc. of the 9th Conference on Computer and Communications Security (CCS)*, pages 255–264, 2002.
37. Xinwen Zhang, J.-P. Seifert, and R. Sandhu. Security enforcement model for distributed usage control. In *Proc. of the Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC)*, pages 10 –18, 2008.

Appendix

Definition 1: (Function set) Let r be a monitored program run. Then the function set of r, which we denote by $\mathbf{functionSet}(r)$, is the smallest set fulfilling the following property: For any invocation $f \rightarrow f'$ of function f' from function f on r, it holds that $\{f, f'\} \subseteq \mathbf{functionSet}(r)$.
Definition 2: (Call graph) A call graph is a directed graph (V, E) with V a set of nodes representing functions, and $E \subseteq V \times V$ a set of directed edges. Then the call graph of r, $\mathbf{cg}(r)$, is a call graph that fulfills the following constraints. V is the smallest set such that for any invocation $f \rightarrow f'$ of function f' from function f on r, it holds that $\{f, f'\} \subseteq V$. E is the smallest subset of $V \times V$ such that for each such f, f' it holds that $(f, f') \in E$.
Definition 3: (Calling context tree) Let F be the set of all function identifiers. Then C_M, the set of all calling contexts over F, is defined as $C_M := F^+$. The set C_M is closed under concatenation: we define a concatenation function "$\cdot$" on calling contexts such that for any context $c \in C_M$ and function $f \in F$ it holds that $c \cdot f \in C_M$. A calling context tree is a tree (V, E) with $V \subseteq C_M$ a set of nodes representing calling contexts and $E \subseteq V \times V$ a parent-child relationship. We further demand that there exists a unique root node v_0 which has no parents, i.e., for which it holds that $\neg\exists v \in V$ s.th. $(v, v_0) \in E$. Let r be a monitored program run. Then $\mathbf{cct}(r)$ is a calling context tree for which the following holds. V is the smallest set such that for any invocation $c \rightarrow f$ of function f from within context c on r, it holds that $\{c, c \cdot f\} \subseteq V$. E is the smallest subset of $V \times V$ such that for each such f, c it holds that $(c, c \cdot f) \in E$.