Monitoring of Security Properties Using BeepBeep

Mohamed Recem Boussaha, Raphaël Khoury, and Sylvain Hallé

Laboratoire d'informatique formelle
Université du Québec à Chicoutimi, Canada

Abstract. Runtime enforcement is an effective method to ensure the compliance of program with user-defined security policies. In this paper we show how the stream event processor tool BeepBeep can be used to monitor the security properties of Java programs. The proposed approach relies on AspectJ to generate a trace capturing the program's runtime behavior. This trace is then processed by BeepBeep, a complex event processing tool that allows complex data-driven policies to be stated and verified with ease. Depending on the result returned by BeepBeep, AspectJ can then be used to halt the execution or take other corrective action. The proposed method offers multiple advantages, notable flexibility in devising and stating expressive user-defined security policies.

1 Introduction

Mobile code has emerged as an effective solution to the challenges of computing in distributed systems. Nonetheless, security concerns remain omnipresent, and may act as a break to the adoption of this technology, in part because of the need for each user to tailor the security policy governing his system to his own need.

In this paper, we show how BeepBeep [5], a complex event processor can be used as a runtime monitor for enforcing a wide array of user-defined security policies. This study also serves to illustrate BeepBeep's capabilities of as a log trace analyzer. BeepBeep takes as input a data stream, in this case an execution trace capturing the method calls and parameters values. This information can be generated using any number of mechanism. BeepBeep has the capacity to efficiently analyse this information in real-time to determine if it conforms with a user defined specification. BeepBeep can also generate useful diagnostic information about the program's runtime behavior, which can in turn be used for further security analysis or debugging.

We rely upon AspectJ [7] to generate the input trace which allows BeepBeep to perform the enforcement. However, the monitoring using BeepBeep is agnostic of the mechanism used to generate the traces, and while AspectJ also exhibits some capabilities to operate as a security policies enforcement mechanism on its own, that is not necessarily the case for other tracers. The use of BeepBeep allows the security enforcement mechanism to be independent from the tracer.

Like other runtime security enforcement mechanism, the approach presented in this paper is *precise*, in the sense that it reject only those executions that

violate the security policy, permitting safe executions of the program to proceed, and it results in no false-positives or false-negatives. It is *late*, in the sense that the execution is not halted until a violation is about to occur, thus allowing as much of the execution to take place as is permissible given the security policy in place. The main advantage of the proposed approach over other monitoring tools is the flexibility and expressivity of the policy specification language.

The remainder of this paper is organised as follows. Section 2 surveys related works. In section 3, we give an overview of the architecture of the security enforcement mechanism proposed in this paper. Section 4 describes some of the security properties we can enforce and Section 5 presents experimental results. Concluding remarks are given in Section 6.

2 Related Work

The Naccio project [4] provides a library of Java security policies that are enforced at runtime. Each policy replaces certain Java Virtual Machine (JVM) classes as needed to allow enforcement, and the JVM must be modified to ensure that the correct (security policy specific) class is used. Any policy part of Naccio's library, as well as a multitude of policies unavailable on that platform, can easily be stated and enforced using BeepBeep.

Several tools leverages machine learning techniques and static analysis to categorize Android applications (written in Java) as either malicious or benign. The tool ANDRANA [3] relies on static analysis of the applications's code to create a vector of features for each application. Classification is then performed to determine if the observed features a typical to those previously observed in malware. Other classifier rely upon the app's manifest file [9], it's service life cycle [6], API calls [1] or a combination of API calls and other statically detected features [2]. Like other methods based on static analysis, these exhibits a risk of false-positive and false-negatives, and are vulnerable to obfuscation.

Another countermeasure in the face of malicious mobile code is the reliance upon of code certification. Code signing utilizes cryptographic keys to guarantee the authorship of code. While a useful security tool, code signing only serves to authenticate the author of a given code, but provides no guarantees as to its actual behavior. The author may be wrongly trusted by a user, and even code from reputable sources can exhibit an exploitable vulnerability.

The approach proposed in this paper is precise and thus risks neither false-positives nor false-negatives. It can be applied to code of unknown origin, and allows the user to easily customize the security policy to his needs. Indeed, as we will show in the next section, it can be used not only to enforce a wide variety of security policies but also to ensure the respect of resource-usage constraints or to generate diagnostic reports about the program's runtime behavior.

3 Architecture

The trace is generated using AspectJ, a tool that allows adding executable blocks to the source code without explicitly changing it. AspectJ allows programmers

to set points in the source, known as pointcuts, to where the execution is to temporarily halt and while the newly added code blocks are executed. In our case, we used AspectJ to insert code before and after every method call to record the information needed to perform monitoring. As mentioned above, this is only one of several methods that could be used to generate the trace.

Figure 1 shows a sample of the trace of a simple Java tutorial program. Each line correspond to either a single method call or method return. The former begin with the keyword the keyword 'call' contains the following information: the method return type, the method's containing class, the method's name, each of the methods parameters type and value, and finally the method's call level on the stack. The later begin with the keyword 'Return' and contain the return value. Values of literals, string and elementary types are provided explicitly but those of objets are provided by references. Arrays are prefixed with '['.

```
Call: return type : void // class: MonitoredProgram // method: main //
type param 1: class [Ljava.lang.String;// value param 1:[Ljava.lang.String;@72ea2f77//level: 0
Call: return type : void // class: java.net.Socket // method: <init> //
type param 1: class java.lang.String // type param 2: int //
value param 1: www.javatutorial.com // value param 2: 80 // level: 1
Return: Socket[addr=www.javatutorial.com/69.172.201.153,port=80,localport=51706]
Call: return type : class java.net.InetAddress //
class: java.net.Socket // method: getInetAddress //  no parameter // level: 1
Return: www.javatutorial.com/69.172.201.153
```

Fig. 1: A fragment of a trace

BeepBeep [5] is a complex event processing tool that can perform complex manipulations on large data streams efficiently. Internally, BeepBeep decomposes the desired data-processing task into a number of atomic *processors*, each of which takes as input one (or more) event streams, and in turn, outputs one or more event streams. These processor are chained together with the output of one (or more) processor being piped to the input of the next one in such a manner that, feeding BeepBeep's input stream through this chain produces the desired computation. Part of the contribution of this paper is to show how complex, data-driven security properties of programs can be stated in terms of a small number of BeepBeep processors.

A benefit of the approach under consideration is the ease with which the desired security property can be stated. Each BeepBeep processors consists in an average 20 lines of Java code, contained in a single class. Users can reuse these elementary blocs, chaining them together to easily compose complex policies.

4 Security Properties

We began by replicating several of the security properties present in Naccio's library. Most of these are safety policies and can be enforced with as little as one or two BeepBeep Processors. Such properties include: NoExec, NoJavaClassLoader, NoNetReceiveing, NoNetSending, NoPrinting, NoReadingFiles, NoListingFiles

LimitBytesWritten and LimitBytesRead, LimitCreatedFiles, LimitObservedFile. The first 7 of these properties simply halt the execution upon encountering a specific forbidden method call. The latter 4 are only slightly more involved. LimitBytesWritten and LimitBytesRead limit to total number of bytes that are written (resp. read) to files or to the network. LimitCreatedFiles and LimitObservedFile limit the number of files that can be created (resp. read).

Figure 2 gives an example of the BeepBeep processors for the property LimitBytesWritten. It consists of only 3 processors: the first extracts from the trace those method calls that perform write operation and passes those method calls to the second processor. The second extracts number of bytes written by from these method calls, and again passes this information on to the next processors. The final processor computes the sum of the values it receives as input and aborts the execution (through AspectJ) if this sum surpasses a customizable value recorded in the property.

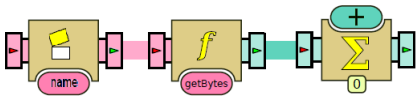

Fig. 2: The BeepBeep processors for property LimitBytesWritten

Schneider introduced the property [10] 'no send after read' as a typical example of a safety property. This property states that after having read from a protected file, the program is no longer allowed to access the network. This property is also part of the Naccio library.

The expressiveness of the approach under consideration is illustrated by the following pair of properties: the property 'a is a key', states that a given piece of information provided in the trace, such a parameter to a specific method or its return value, never take the same value twice. Conversely, it's negation, the property 'a is not a key' requires that this value be unique. These properties are interesting since, as observed by Segoufin [11], several widely used data models can express one of these properties, but not the other, and neither of these properties is part of the Naccio library. Both can be stated using relatively simple processors, that stores the values that have appeared so far in the execution in a list, and consult this list before allowing the execution to proceed.

Deserialization attacks [8] have recently emerged as important vector of attack against Java programs. Any program that relies upon serialized objects to exchange information with a distant party may be susceptible to a serialization attack, even if the data is validated after having been received. The attack can be performed in any one of several ways, notably by sending data of an unexpected type, by sending an object of the correct type but with a high order nested structure, leading to resource exhaustion when it is deserialized or by sending an object whose fields values are not consistent with the normal execution of the program. Beep Beep processors offer a simple and effective counter-measure in all cases. Since the trace contains return values and their type, validation can

be performed with a processor similar to the ones described above. To protect against a deserialization bomb, we developed a processor that bounds the number of consecutive nested calls of the `"readObject()"` method. BeepBeep can also ensure important data secrecy properties by preventing data read in sensitive files from being included in the serialized object.

BeepBeep allows us to state more complex policies that relate the values present in different parts of the trace to one another. For example:

- The parameter values of a given function are always increasing/decreasing in consecutive calls. This property ensures the correctness of recursive function.
- After being created, a given data object is not modified (data integrity).
- No thread is frozen for more than 100 milliseconds before resuming its execution (starvation freedom).
- Whether two specific methods work on the same object, or alternatively provide a list of objects that are manipulated by both of these methods.

Since the output of a processor can be of any format, BeepBeep can also provide profiling information about the ongoing execution, such as maximal, minimal and average stacks depth, the number of objects created for each object type, etc. We present two final processors that illustrate this capability of BeepBeep: processor `"CallSequenceProfiling"` lists, for each method call in the trace, the number of times it directly calls every other method. This information is provided in the form of a directed weighed graph, in which each vertex is labelled with a method name, and a vertex of weight c is present between vertexes v_1 and v_2 iff method v_1 calls method v_2 c times in the trace. We give a schematic representation of this processor in Figure 3. The processor `"BytesWrittenGraph"` provides a list of every method that manipulates each data object. Recall that data objects are identified by reference in the trace. These two processors, while not security property enforcers, provide crucial information on data flow analysis that is essential to debugging and to the enforcement of data flow policies.

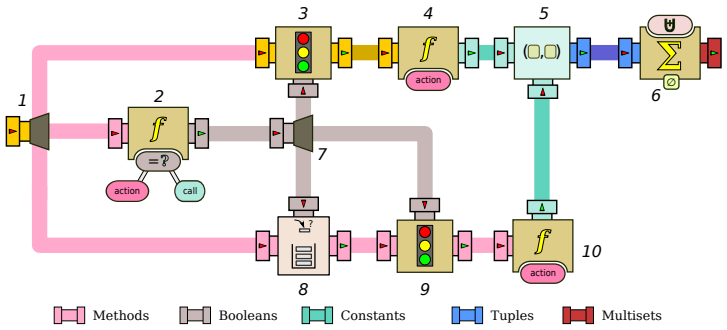

Fig. 3: The BeepBeep processor chain for property CallSequenceProfiling

Figure 3 shows the chain of processors required to compute the call graph from an execution trace. The core of this chain is the `Stack` processor, depicted as

number 8 in the figure. It takes two event streams as its input: the first (left-hand side) is a stream of events of an arbitrary type; the second (top side) is a stream of Boolean values. This processor internally maintains a stack of received events. To this end, the Boolean stream acts as a *push flag*. When an event e and a Boolean value b arrive at the processor's inputs, two situations may occur. If $b = \top$, the top of the stack (if not empty) is output but not removed, and e is then pushed onto the internal stack if $b = \top$. If $b = \bot$, e is ignored, and the element at the top of the internal stack is popped and discarded.

The original stream of method events is first split in three (1); one of these copies is given as the input to the `Stack` processor (8), while another is sent to a `Function` processor (2). This processor evaluates the function that compares the `action` field of the method event with the constant `call`; the result is a stream of Boolean values, indicating whether the incoming event is a method call ($\top$) or a method return ($\bot$). This stream itself is split in three (7), and one of these copies is given as the push flag of the `Stack` processor. The stack is hence instructed to push an incoming event when it is a method call, and to pop the top of the stack when it is a method return. As a result, the output of the `Stack` processor is the method event corresponding to the current method in the program's execution.

A third copy of the original stream of events is sent to a `Filter` processor (3). This processor receives two inputs: an arbitrary event e and a Boolean value b called the *filter flag*. Event e is output if $b = \top$, otherwise e is discarded. The filter flag, in this case, is the result of applying the function `action = call`, which returns a Boolean value; in other words, the processor keeps only method call events, and filters out method returns. The same filtering condition is applied to the output of the `Stack` processor (9).

The end result of this first part of the chain is that processors 3 and 9 synchronously output method call events; events at matching positions in the streams represent the caller of a method (9) and the method being called (3). From this point on, the rest of the processing is straightforward. Both events are processed so that only the value of their `name` field is kept (4, 10); these two values are then joined in a tuple (5), and these tuples are then accumulated into a multiset using a `CumulativeProcessor` (6).

The output stream resulting from 6 is a sequence of multisets, each of the form $\{(m_0, n_0), \ldots, (m_k, n_k)\}$; each tuple (m_i, n_i) is a caller-callee pair of method names. The number of times each distinct pair occurs in the multiset corresponds to the number of times n_i was called from m_i in the trace. From then on, it is easy to take the multiset of tuples and covert it into a directed graph that shows the weighted dependencies between methods in the observed execution.

It is worth mentioning that, in this whole graph, only the `action` function (used in processors 2, 4 and 10) and `Stack` processor (8) are specific to our use case. This amounts to less than 150 lines of custom code. All the remaining processors and functions are generic, and already come in BeepBeep's core or one of its existing palettes.

Figure 4 shows a more informative variant of the `LimitBytesWritten` property. It computes the number of bytes written by each function that does so, and expresses this information in the form of a plot. The processor chain begins by

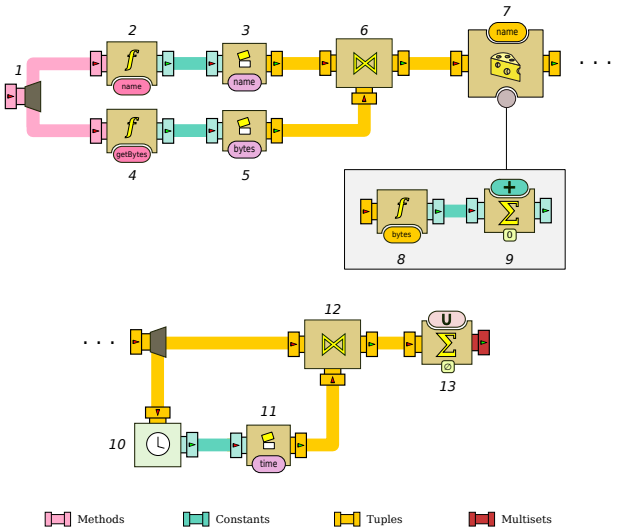

Fig. 4: The BeepBeep processor chain for property BytesWrittenGraph

extracting from the trace the methods that perform write operations, discarding all other lines and pairs containing the method name and the number of bytes written are joined in a tuple (processor 6). Processor 7 splits its input stream into multiple distinct streams, each of which contains tuples originating from a single method. This allows the computation of the number of bytes written to be aggregated separately for each method (processors 9 and 10). The remainder of the processor chain aggregates this information with a timestamp, and updates a hash table accordingly. This hash table can then serve as the basis of a plot generated on demand.

5 Experimental Results

We tested this method on traces of length 1 000 000, generated in the manner described above on a Java calculator. Figure 5 plots the execution times (in ms.) for four representative processor chains, namely `NoExec`, `LimitBytesWritten`, `CallSequenceProfiling` and `isAKey`. As can be seen in these results, execution times are largely proportional to the number of processors in each processor chain. Since most security properties require only linear sequencing of processors, their operation can easily be streamlined by merging the operations of multiple monitor in a single class.

Figure 1 details the number of processors, number of custom lines of code (not counting code already present in BeepBeep's template library) and execution time several of the processors mentioned in this paper. This table illustrates the ease with with BeepBeep processors can be composed, often necessitating only a minimal amount of custom code. Once these processor chains are implemented,

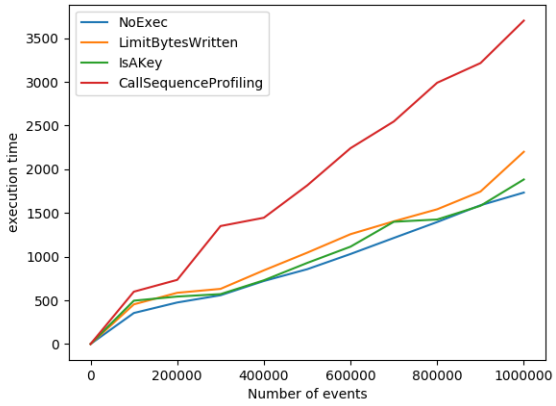

Fig. 5: Experimental Results

they can in turn be included as components of processor chains for more complex properties with the addition of a single line of code.

The only processor chain whose execution time is not inconsiderable is `BytesWrittenGraph`, described in the previous section. Much of its execution time is incurred in the final processor, which updates a hash table linking each method to the number of bytes that have been written by that method during the program's current execution. Since a BeepBeep processor chain manipulates events sequentially, and each processor feeds its output to a successor, the final processor of the `BytesWrittenGraph` processor chain performs the update by copying the current hash table before updating it with the information it has received in its last input event. However, since in our particular case, the hash table update is performed in the final processor of the processor chain. As a result, it is possible to edit this processor so that is simply updates the hash table, without making a copy. This revision brings `BytesWrittenGraph`'s execution time in line with that of other monitors of its size.

6 Conclusion

In this paper, we showed how the event processor BeepBeep can be used for runtime enforcement. The approach is agnostic to the tracer used to generate the trac and itself by the ease by which properties can be stated using BeepBeep's processor chain structure. BeepBeep is useful not only for stating and enforcing security properties, but also ro generate useful diagnostic information about the trace, as we also illustrated using examples. One avenue of further research which we are currently exploring is to draw on BeepBeep's capabilities to allow us to express a more informative verdict that simply a boolean indication of the respect\violation of the security property. For instance, the monitor could provide indications as to which parts of the trace caused the violations, rate its severity, and suggest weaker properties that are respected. This information

Property	nb of processors	size (lines)	exec. time (ms)
NoExec	2	6	1590
NoClassLoader	2	6	1636
NoNetwork	2	6	1654
NoReadingFiles	2	6	1699
IsKey	2	8	1883
LimitBytesWritten	3	11	1900
CallSequenceProfiling	8	35	3701
BytesWrittenGraph	13	53	14633

Table 1: Description of tested Monitors

could, in turn, serve as the basis for a more corrective reaction to a potential violation than simply aborting the execution.

References

1. Aafer, Y., Du, W., Yin, H.: DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android, pp. 86–103. Springer International Publishing, Cham (2013), http://dx.doi.org/10.1007/978-3-319-04283-1_6
2. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: Drebin: Effective and explainable detection of android malware in your pocket. In: NDSS. The Internet Society (2014)
3. Bedford, A., Garvin, S., Desharnais, J., Tawbi, N., Ajakan, H., Audet, F., Lebel, B.: Andrana: Quick and accurate malware detection for android. In: Foundations and Practice of Security - 9th International Symposium, FPS, Québec City, QC, Canada, October 24-25, 2016,. pp. 20–35 (2016)
4. Evans, D., Twyman, A.: Flexible policy-directed code safety. In: 1999 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 9-12, 1999. pp. 32–45. IEEE Computer Society (1999), https://doi.org/10.1109/SECPRI.1999.766716
5. Hallé, S.: When RV meets CEP. In: Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30. pp. 68–91 (2016)
6. Khanmohammadi, K., Rejali, M.R., Hamou-Lhadj, A.: Understanding the service life cycle of android apps: An exploratory study. In: Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices. pp. 81–86. SPSM '15, ACM, New York, NY, USA (2015)
7. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ, pp. 327–354. Springer Berlin Heidelberg (2001)
8. Lai, C., Microsystems, S.: Java insecurity: Accounting for subtleties that can compromise code. IEEE Software (2008)
9. Sato, R., Chiba, D., Goto, S.: Detecting android malware by analyzing manifest files. In: Proceedings of the Asia-Pacific Advanced Network
10. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (2000), http://doi.acm.org/10.1145/353323.353382
11. Segoufin, L.: Automata and logics for words and trees over an infinite alphabet. In: Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, Sept. 25-29,. pp. 41–57 (2006)