Execution Trace Analysis Using LTL-FO$^+$

Raphaël Khoury, Sylvain Hallé, and Omar Waldmann

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

Abstract. We explore of use of the tool BeepBeep, a monitor for the temporal logic LTL-FO$^+$, in interpreting assembly traces, focusing on security-related applications. LTL-FO+ is an extension of LTL, which includes first order quantification. We show that LTL-FO$^+$ is a sufficiently expressive formalism to state a number of interesting program behaviors, and demonstrate experimentally that BeepBeep can efficiently verify the validity of the properties on assembly traces in tractable time.

1 Introduction

Traces are a useful basis for several types of program analysis when the source code may be unavailable, including notably debugging [7], performance analysis [1], and feature enhancement. Assembly traces are particularly interesting since they provide a detailed picture of the target program's runtime behavior, and can thus be used to detect the occurrence of behaviors that would otherwise not be observable if we relied only on higher level information traces such as system calls or program functions calls. However, the large volume of information present in assembly traces raises several challenges, especially with regards to scalability and pattern detection.

Assembly traces have often been eschewed by security researchers, perhaps because of tractability problems related to their size. Recently, Roşu et al. [8] proposed an extension of past time LTL in which matching function calls and returns are explicitly stated. Ghiasi et al. [4] propose a method that relies upon the register values at the moment when critical library functions are called and return to detect a wide variety of malware.

In this study, we propose a new approach for the verification of assembly traces through runtime monitoring. Security properties are expressed in a first-order extension of Linear Temporal Logic, called LTL-FO$^+$, and are verified using the BeepBeep monitor [5]. We focus on the detection of potential error conditions in the trace, call sequence profiling as well as the detection of specific, potentially malicious patterns in the trace. What distinguishes our study from previous ones is the ease with which properties of any kind can be stated over the relationship between any values present in the trace. The properties we check encompass that of previous related work, and as we will show, most of them could not have been stated in a formalism less expressive than LTL-FO$^+$.

Previous work on enforcing security policies on assembly traces focused on developing heuristics to allows the detection of malware. For example, Ghiasi et al. [4] extract from the trace a model that captures the target program's register values before and

after each system call, and raises an alarm if the observed values diverge sufficiently from expected values. Likewise, Storlie et al. [10] build a Markov chain model from the program's assembly trace. While these methods have high accuracy rate (in the 90%-98% range), we believe we are the first to capture the specific behavior of malware using temporal logic and detecting it over assembly traces. This detection method is even more accurate and avoid the possibility of false positives. Our study further shows that LTL-FO$^+$ is sufficiently expressive to state the properties of interest and that the BeepBeep monitor is capable of handling the considerable size of assembly traces.

2 First-Order Linear Temporal Logic

The language we present in this section is LTL-FO$^+$, a first-order extension of a well-known logic called Linear Temporal Logic (LTL), which has been shown to be appropriate for the modelling of so-called "data-aware" properties [5].

The building blocks for asserting properties over event traces are *atomic propositions*, which are of the form $x = y$, where x and y are either variables or constants. These atomic propositions can then be combined with Boolean operators $\wedge$ ("and"), $\vee$ ("or"), $\neg$ ("not") and $\rightarrow$ ("implies"), following their classical meaning. In addition, LTL *temporal operators* can be used. The temporal operator **G** means "globally"; hence, the formula $\mathbf{G}\,\varphi$ means that formula φ is true in every event of the trace, starting from the current event. The operator **F** means "eventually"; the formula $\mathbf{F}\,\varphi$ is true if φ holds for some future event of the trace. The operator **X** means "next"; it is true whenever φ holds in the next event of the trace. Finally, the **U** operator means "until"; the formula $\varphi\,\mathbf{U}\,\psi$ is true if φ holds for all events until some event satisfies ψ.

Finally, LTL-FO$^+$ adds *quantifiers* that refer to parameter values inside events. Formally, the expression $\exists x \in \pi : \varphi(x)$ states that in the current event m, there exists a value $v \in m(\pi)$ such that $\varphi(v)$ is true. The notation $m(\pi)$ denotes the set of element values in m matching some pattern π. Dually, the expression $\forall x \in \pi : \varphi(x)$ requires that $\varphi(v)$ holds for all $v \in m(\pi)$. When the context is clear, we abbreviate $\exists x \in \pi : x = k$ as π/k, stating that the (only) value at the end of path π is k. The semantics of LTL-FO$^+$ are summarized in Table 1. The symbol $\overline{m}$ represents the execution trace while $\models$ indicates that the expression before the symbol is a model of the one following it : the first part satisfies the following expression. The symbols $\overline{m}_0$ and $\overline{m}_1$ represent the first and second elements in the trace while $\overline{m}^1$ represents the second event and the ones following it, basically the trace without the first element.

The path expression π is a slash-separated list of elements, which denotes a sequence of nested elements in an XML structure; for example the expression `/a/b/c` denotes the value of all elements named `c` inside some `b` element, itself inside some `a` element at the root of the document. The double slash looks for all (and not only immediate) descendants of given name. Finally, the square brackets denote predicates, i.e. conditions that must apply to the element to be included in the selection. For example the path `//a[c=1]/b` fetches the value of all `b` elements nested in some `a` element, itself containing an element `c` with value 1.

$$
\begin{aligned}
\overline{m} \models x = y &\equiv x \text{ equals } y \\
\overline{m} \models \neg\varphi &\equiv \overline{m} \not\models \varphi \\
\overline{m} \models \varphi \wedge \psi &\equiv \overline{m} \models \varphi \text{ and } \overline{m} \models \psi \\
\overline{m} \models \varphi \rightarrow \psi &\equiv \overline{m} \not\models \varphi \text{ or } \overline{m} \models \psi \\
\overline{m} \models \mathbf{G}\,\varphi &\equiv m_0 \models \varphi \text{ and } \overline{m}^1 \models \mathbf{G}\,\varphi \\
\overline{m} \models \mathbf{X}\,\varphi &\equiv \overline{m}_1 \models \varphi \\
\overline{m} \models \varphi\,\mathbf{U}\,\psi &\equiv \overline{m}_0 \models \psi, \text{ or both } \overline{m}_0 \models \varphi \text{ and } \overline{m}^1 \models \varphi\,\mathbf{U}\,\psi \\
\overline{m} \models \forall x \in \pi : \varphi &\equiv \overline{m} \models \varphi[x/v] \text{ for all } v \in m_0(\pi)
\end{aligned}
$$

Fig. 1: Semantics of LTL-FO$^+$

3 Temporal Properties over Assembly Traces

In this section, we give examples of some of the type of properties that we have been able to verify by applying LTL-FO$^+$ to assembly traces. The traces we use are assembly-level generated by a proprietary tracer for Intel 64 developed at the center of Research and Development for Defence Canada (RDDC) based in Valcartier, Québec. Each line of these files corresponds to a single assembly-level instruction performed by the system being monitored or to a system call. While none of the properties used in this paper contains a system call, they are present in the trace and could be a component of the property.

```
06bb5c mov esp, ebp | EBP=001bfbf4 | ESP=001bfbf4
06bb5d pop ebp | ESP=001bfbf4 [001bfbf4]=001bfc24 | EBP=001bfc24 ESP=001bfbf8
06bb5e push ecx | ECX=71f1a8b9 ESP=001bfbf8 | ESP=001bfbf4 [001bfbf4]=71f1a8b9
06bb5f ret | ESP=001bfbf4 [001bfbf4]=71f1a8b9 | ESP=001bfbf8
06bb60 ret | ESP=001bfbf8 [001bfbf8]=01391036 | ESP=001bfbfc
06bb61 add esp, 0x20 | ESP=001bfbfc | ESP=001bfc1c EFLAGS=
06bb62 cmp  [ebp-0x4], 0x3e8 | [001bfc20]=000003e8 EBP=001bfc24 | EFLAGS=ZP
06bb63 jnz 0x1391057 | EFLAGS=ZP |
06bb64 push 0x1392144 | ESP=001bfc1c | ESP=001bfc18 [001bfc18]=01392144
```

Fig. 2: A fragment of assembly trace

A short fragment of a trace is shown in Fig. 2. Each line begins with a sequential number, and then contains three sets of information, separated by |. The first is the assembly instruction that occurred, with its parameters. This is followed by the initial values present in each relevant register or memory location before this instruction is executed. Finally, the tracer records the values of registers, memory locations or flags that were modified by the instruction. In this example, the trace is that of a simple C program that reads an integer value from a file, checks whether or not the value that

has been read equals 1000 (which is the case in our example) and branches accordingly. The relevant assembly instructions are on lines 06bb62 and 06bb63 of the trace file. The complete trace file contains about 470,000 lines.

We focus on five types of properties of assembly traces, which we describe below. These examples do not exhaust the possible uses of temporal logic verification over assembly traces. Rather, they illustrate the diversity of behavior that can be stated in LTL-FO$^+$ and verified over assembly traces using BeepBeep. The properties are reproduced in the Appendix.

Property 1: Integer overflow detection Integer overflow is a potential source of abnormal program behavior if its occurrence was unanticipated by the program developer or if it is unsupported or undefined by the programming environment. For example, integer overflow of signed integers is undefined behavior in C and C++ , despite being widely relayed upon by programmers [3]. Integer overflow is also a know source of vulnerabilities in programs. For example, the "jpeg of death" vulnerability was caused by an integer overflow [6].

Property 2: Call sequence profiling is a program analysis that serves to reconstitute the call-chain occurring in the target program during its execution. It has applications notably in program comprehension, software maintenance and software re-engineering, amongst other. In this paper, we consider two properties, namely determining if it is possible for a given function to be called by a specific caller, and if a caller function always calls a given callee before returning.

Property 3: Return address protection One of the main classes of malicious code functions by overriding the return address on the stack before the executing function returns. Buffer overflow vulnerabilities, return-to-LibC and return oriented programming are all examples or malware that exploit this type of vulnerability.

Property 4: Pointer subterfuge detection A pointer subterfuge is a potentially malicious behavior in which a value is written to memory through a buffer-copy or a string manipulation operation, and then used as the target of a function call [9].

Property 5: Malicious Pattern detection Christodorescu et al. [2] showed how logical expressions can be used to describe specific malicious patterns of assembly code. Such patterns can be used to detect the presence of malicious code even when obfuscation techniques have been used to hide their presence. In our final example we wrote an LTL-FO$^+$ formula that detects the presence of this pattern in the trace.

It is important to stress that these properties simply could not have been stated in a formalism less expressive than LTL-FO$^+$, which includes quantified variables over complex events. This is because the interpretation of the property requires that we reason about the relationship between the different elements present in the trace. For instance, to determine if a return address on the stack is overwritten, we check that whenever a call event occurs, the return address it places on the stack is not overwritten until the occurrence of the *corresponding* ret instruction (regardless of how many call-ret pairs occur in the interim). In this absence of quantified variable in the formula, the same effect could only be achieved by including in the formula every possible pairing of call-ret values—in our case, every virtual memory location in use in the execution environment. The size of the automaton representing this property would be exponential in the number of such locations.

Other properties would be even harder to state. Pointer subterfuge detection involves verifying if a given write, occurring in a buffer, is later used as the destination for of a call. A simple LTL-FO$^+$ formula achieves this by quantifying over the location and value of `mov` instruction, and then verifying that these same values never occur as the location and destination of a call instruction. (see property in Appendix 1). A flag indicating that the write instruction occurred as part of a buffer write (the `rep` flag was set) helps constraint the search for optimization purposes. Absent quantified variable int he formula, it is not clear how such a property could be checked, short checking every `mov` instruction in the trace against every single call instruction (Up to 30% of all events occurring in each sample trace where `mov`s).

4 Experimental Results

We tested the five properties described above using traces of varying length, up to 500,000 assembly instructions. A preprocessing script was run over each trace file beforehand in order to format them into an XML structure suitable for verification with BeepBeep. The tests were generated using the ParkBench test suite.[1] The results are summarized in Table 1.

	Number of lines	Number of quant.	Number of connectives	Nesting Depth	Total exec. time (ms)	Max Heap (MB)	Avg time per event (ms)
Property 1	500,000	0	2	2	175611	593	0.35
Property 2	500,000	1	9	8	6245	481	0.12
Property 3	500,000	2	6	7	30996224	1331	61.99
Property 4	500,000	2	6	8	6360461	739	12.72
Property 5	381,583	5	20	19	293033440	1969	767.94

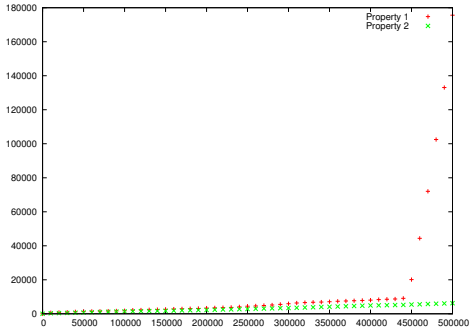
Table 1: Experimental results

Fig. 3a and 3b show the detailed execution time of the monitor for each property. With the exception of Property 5, the execution times grow linearly with the size of the execution trace at a very tractable rate. Properties 1 and 3 both exhibit sudden increases in their execution time. This seems to be related to the functioning of the Java garbage collection. The execution time is generally proportional to the size of the size of the property under consideration. Fig. 4 shows the heap usage for each property. Most properties require a constant amount of memory regardless of of the size of the execution trace.

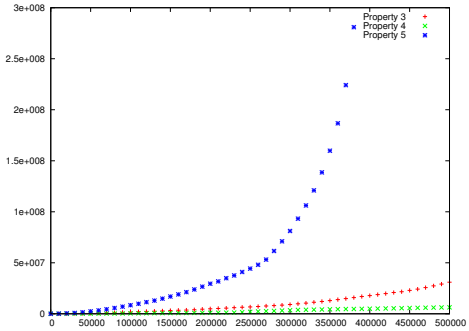
5 Conclusion

In this paper, we showed that the BeepBeep monitor is an effective tool to verify LTL-FO$^+$ properties over assembly traces. The expressive power of LTL-FO$^+$ permits users

[1] `http://github.com/sylvainhalle/ParkBench`



(a) Prop. 1–2



(b) Prop. 3–5

Fig. 3: Experimental results: Execution time

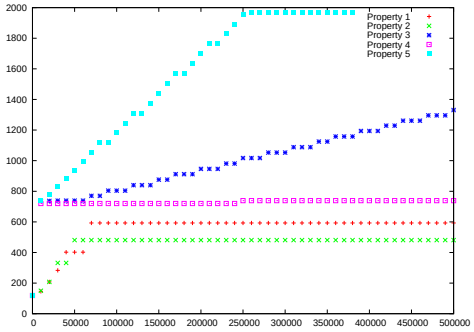


Fig. 4: Experimental results: Heap usage

to state fine-grained properties, drawing on the specific values present in the trace, thus allows users to harness the detailed content of assembly-level traces. Furthermore, we showed that the BeepBeep monitor is sufficiently powerful to verify these properties on real traces, despite their considerable size.

We are currently exploring other uses of LTL-FO$^+$ and of the BeepBeep monitor. The LTL-FO$^+$ language is sufficiently expressive to state interesting properties for several types of program traces, in addition tot he assembly-level traces discussed in this paper. Likewise, the BeepBeep monitor is capable of verifying such properties in tractable time over traces of considerable size. We are especially interested in applying these tools to the problems associated with program comprehension and program re-engineering, which require a thorough understanding of method call and system call traces.

References

1. D. Becker, F. Wolf, W. Frings, M. Geimer, B. Wylie, and B. Mohr. Automatic trace-based performance analysis of metacomputing applications. In *Parallel and Distributed Processing*

Symposium, 2007, pages 1–10, March 2007.
2. M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, SP '05, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.
3. W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 760–770, Piscataway, NJ, USA, 2012. IEEE Press.
4. M. Ghiasi, A. Sami, and Z. Salehi. Dynamic VSA: a framework for malware detection based on register contents. *Engineering Applications of Artificial Intelligence*, 44:111–122, 2015.
5. S. Hallé and R. Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Trans. Services Computing*, 5(2):192–206, 2012.
6. C. Hornat. Jpeg vulnerability: A day in the life of the jpeg vulnerability. Technical report, Info Security Writers, 2004.
7. D. Jerding and J. Stasko. The information mural: a technique for displaying and navigating large information spaces. *IEEE Transactions onVisualization and Computer Graphics*, 4(3):257–271, Jul 1998.
8. G. Rosu, F. Chen, and T. Ball. Synthesizing monitors for safety properties: This time with calls and returns. In M. Leucker, editor, *Runtime Verification, 8th International Workshop, RV 2008, Budapest, Hungary, March 30, 2008. Selected Papers*, volume 5289 of *Lecture Notes in Computer Science*, pages 51–68. Springer, 2008.
9. R. C. Seacord. *Secure Coding in C and C++*. Addison-Wesley Professional, 2nd edition, 2013.
10. C. Storlie, B. Anderson, S. Vander Wiel, D. Quist, C. Hash, and N. Brown. Stochastic identification of malware with dynamic traces. *Annals Of Applied Statistics*, 8(1):1–18, 2014.

Appendix: LTL-FO$^+$ Properties

Property 1: Integer Overflow Detection

$$\mathbf{G}\,((\text{instruction} = \mathit{add}) \rightarrow (\text{overflow-flag} = \text{false}))$$

Property 2: Call Sequence Profiling

$$\begin{aligned}&\mathbf{F}\,(\exists \mathit{Address}1 \in \text{return-address} : ((((\text{instruction} = \text{call})\\ &\wedge (\text{destination} = 71f1acdc)) \wedge (\neg((\text{instruction} = \mathit{ret}) \wedge (\mathbf{X}\,(\text{location} = \mathit{Address}1)))))\\ &\qquad\qquad \mathbf{U}\,((\text{instruction} = \text{call}) \wedge (\text{destination} = 71f1a42e))))\end{aligned}$$

Property 3: Return Address Protection

$$\begin{aligned}&\mathbf{G}\,((\text{instruction} = \text{call}) \rightarrow (\forall \mathit{eip} \in \text{locOnStack} :\\ &(\forall \mathit{retAddrVal} \in \text{return-address} : ((\neg((\text{instruction} = \mathit{mov}) \wedge (\mathit{output}/\mathit{value} = \mathit{eip})))\\ &\qquad\qquad \mathbf{U}\,((\text{instruction} = \text{return}) \wedge (\text{function-returned} = \mathit{retAddrVal}))))))\end{aligned}$$

Property 4: Pointer Subterfuge Detection

$$\begin{aligned}
&\mathbf{G}\,((\text{rep-flag} = \text{True}) \rightarrow (\forall addr \in \text{destination} : \\
&\qquad (\forall val \in \text{value} : (\neg(\mathbf{F}\,((\text{instruction} = \text{call}) \wedge \\
&\qquad\qquad ((locOnStack = addr) \wedge (\text{destination} = val))))))))
\end{aligned}$$

Property 5: Malicious Pattern Detection

$$\begin{aligned}
&\mathbf{G}\,(\exists retAddrVal \in \text{return-address} : ((\text{instruction} = \text{call}) \wedge (\neg((\mathbf{F}\,(((\text{instruction} = mov) \\
&\qquad \wedge (\text{output/type} = \text{general-register})) \rightarrow (\exists regA \in \text{output/name} : \\
&(\mathbf{F}\,((((\text{instruction} = mov) \wedge (\text{output/type} = \text{general-register})) \wedge (\text{input/type} = literal)) \rightarrow \\
&\qquad (\exists regB \in \text{output/name} : (\exists constAddr \in \text{input/value} : (\mathbf{F}\,(((\text{instruction} = \text{cmp}) \wedge \\
&\qquad\qquad (\text{output/type} = regA)) \rightarrow (\exists loc \in \text{location} : (\mathbf{F}\,((((\text{instruction} = mov) \wedge \\
&\qquad\qquad\qquad (\text{output/type} = \text{general-register})) \wedge (\text{output/name} = regA)) \wedge \\
&\qquad\qquad\qquad ((\text{input/name} = regB) \wedge (\text{input/type} = ptr)))))))))))))) \\
&\qquad\qquad\qquad \mathbf{U}\,((\text{instruction} = return) \wedge (\text{function-returned} = retAddrVal))))))
\end{aligned}$$