Optimized Inlining of Runtime Monitors

Frédérick Lemay[1], Raphaël Khoury[1,2], and Nadia Tawbi[1]

[1] Laval University, Department of Computer Science and Software Engineering,
Pavillon Adrien-Pouliot, 1065, avenue de la Medecine Quebec, Qc, Canada G1V 0A6, Canada
[2] Defence Research and Development Canada, Canadian Department of National Defence
2459 Pie-XI Blvd North Quebec, QC, Canada G3J 1X5
frederick.lemay.1,raphael.khoury.1@ulaval.ca,nadia.tawbi@ift.ulaval.ca

Abstract. A previous study showed how a monitor can be inlined into a potentially untrusted program, producing an instrumented version of this program which provably respects the desired security policy. That study extended previous approaches to the same problem in that it allowed non-safety properties to be monitored, and did not incur any runtime overhead. However, the algorithm itself runs in time $\mathcal{O}(2^{m \cdot n})$, where n is the size of the original program and m that of the property being monitored, and the resulting instrumented program is increased in the order of $\mathcal{O}(m \cdot n)$. These algorithmic factors limit the usefulness of the approach in practice. In this paper, we suggest several optimizations which reduce the algorithm's run time and the size of the resulting instrumented code. Using these optimizations, the monitor inlining can run in time $\mathcal{O}(v + e)$ where v and e are respectively the size and number of transitions present in the synchronous product of the original program and the property. Furthermore, we show how the size of the instrumented program can be minimized.

Key words: security policies, security properties, omega-automata, runtime monitors

1 Introduction

An increasingly popular solution to the problem of securing mobile code is monitor inlining, a process by which a monitor, representing a security property, is injected into an untrusted program. This results in a new instrumented version of the program which provably respects the desired property.

Much research has gone into determining precisely which properties are enforceable using this approach, and which are not. Initial research showed that this method was limited to the enforcement of safety properties, and implementations of formal monitors naturally focused on this class of security properties. Yet, further studies on security policy enforcement mechanisms showed that an a priori knowledge of the target program's behavior increases the power of these mechanisms [21, 6].

Nonetheless, most practical implementations of monitors do not take advantage of this possibility and are thus restricted to enforcing safety properties. Furthermore the needed abstraction is often already available, and is used to minimize the runtime overhead incurred by the property monitoring process.

In [11, 10] Chabot et al. present a monitor inlining algorithm in which the monitor relies on a model of the program's possible behavior to enforce non-safety properties. While their method is strictly more expressive than previous similar monitor inlining algorithms and the instrumentation induces no added runtime overhead, the very high computational complexity of the algorithm limits the applicability of the method.

In this paper, we propose several optimizations to the original monitor inlining algorithm from [11]. These allow the algorithmic complexity to pass from $\mathcal{O}(2^{n \cdot m})$ for a program of size n and a property of size m to a much more tractable $\mathcal{O}(v + e)$, where v and e are respectively the number of states and transitions present in the synchronous product of the original program and the desired property. We also show how to minimize the size of the resulting instrumented code, which stood at $\mathcal{O}(n \cdot m)$ in the original approach.

These optimizations derive from two insights: first, using an alternative formalism to express the desired security policy can speedup certain computations, specifically cycle detection, which is the most time consuming aspect of the algorithm. Second, the target program, when abstracted into a graph, exhibits specific properties that can be used to our advantage throughout the transformation process. Empirical results show that we can successfully reduce the time needed to perform the program transformation.

The remainder of this paper is organized as follows: Section 2 presents a review of related work, Section 3 introduces some preliminary notions and Section 4 presents the monitor inlining algorithm from [11]. Section 5 gives the proposed optimizations. Empirical results are given in Section 6. Concluding remarks are given in Section 7.

2 Related Works

In [38], Schneider delineates the set of properties enforceable by monitors. He focuses on a class of monitors that enforce the property by aborting the execution when faced with a violation of the security policy, and operate without any prior knowledge of their target's possible behavior. In these conditions, he determined that the set of security properties enforceable by this mechanism is the set of safety properties [25], a class of properties which proscribe that a certain unwanted behavior will not occur during a given execution of the target program. However, he also suggests that non-safety properties can be monitored in several situations, for example if the monitor had access to a model of its target possible behavior, which can be constructed from data collected by static analysis. In this case, the monitor can tolerate a potentially invalid behavior on the part of the target program, with the confidence that it will eventually be corrected. Subsequent research by Ligatti et al. [6] confirmed the feasibility of extending the set of monitorable properties using such an enforcement paradigm.

Several subsequent studies show how monitors can be inlined into their target, thus producing a new version of an untrusted program that provably respects the desired security policy. For instance, in [16], Schneider et al. proposed a method to inline a security property in object code, while Colcombet et al. [13] propose a similar approach which seeks to secure source code. In both cases, the method is limited to the enforcement of safety properties. The inlining process involves synchronizing the program with the security property by way of runtime checks that simulate the behavior of the secu-

rity automaton into the target program. These checks detect any violation of the security property at runtime, and abort the execution. Both Schneider et al. and Colcombet et al. propose a number of optimizations which minimize the number of runtime checks needed to enforce the property.

In [10, 11], Chabot et al. extend these approaches by allowing the monitoring of non-safety properties. Drawing upon the insight of [38] and [6], they propose a monitor which relies upon a statically constructed model of the target to enforce non-safety properties in some cases. While their approach is strictly more expressive than previous the ones, its algorithmic complexity, which stands in the order of $\mathcal{O}(2^{n \cdot m})$ for a program of size n and a property of size m, is a major hindrance to its wider usage.

Numerous other implementations of inlined monitors exist. For instance, in [34, 33] and [35], Ould-Slimane et al. give an automata based inlining procedure which relies on a new operator that embeds a property automata into a target program. The monitoring of security protocols is discussed in [4] and [5]. That of information flow policies is discussed in numerous papers including [12] and [17]. In [39] Sen at al. propose a decentralized monitor which monitors safety properties in distributed programs. The optimization of monitors is further discussed in [44]. The inlining of monitors in concurrent programs is discussed in [27]. An algebraic method to inline a safety property into a program is given in [26] and [28]. In this approach, both the property and the program are stated using process algebra. The instrumented program is shown to be equivalent to the original one using a notion of equivalence based on bisimulation. The monitoring of networks is discussed in [31].

The monitoring of nonsafety properties is also discussed in [29], which shows that such properties can be monitored by an enforcement mechanism capable of transforming the execution sequence.

3 Preliminaries

We begin by introducing in more detail the notation and the various types of ω-automata which we will manipulate in this paper, and explain how such automata accept or reject their input.

An alphabet is a finite non-empty set of symbols. A word over alphabet Σ is a sequence of symbols from Σ. In what follows Σ^* denotes the set of all finite words from Σ while Σ^ω denotes that of all infinite words. A language $\mathcal{L}$ is a subset of Σ^ω and/or Σ^*. Security properties are also modeled as subsets of Σ^ω and automata are a convenient formalism to represent them [2].

An ω-automaton $\mathcal{A}$, over alphabet Σ is a tuple $(Q, \mathcal{I}, \delta, C)$ such that

- Σ is a finite or countably infinite set of symbols;
- Q is a finite or countably infinite set of states;
- $\mathcal{I} \subseteq Q$ is the set of initial states;
- $\delta \subseteq Q \times \Sigma \times Q$ is a (possibly partial) transition function;
- C is an acceptance condition which specifies whether or not an infinite sequence ρ is accepted by the automaton as depending on the states which occur or do not occur infinitely often in ρ. This condition is stated differently in the various types of ω-automata which we will study, sometimes leading to variations in expressive

power. Accepted sequences are said to be valid, while rejected sequences are said to be invalid.

The set of all accepted sequences of $\mathcal{A}$ is the language recognized by $\mathcal{A}$, noted $\mathcal{L}_{\mathcal{A}}$.

A *path* ρ, is a finite (respectively infinite) sequence of states $\langle q_1, q_2, ..., q_n \rangle$ (respectively $\langle q_1, q_2, ... \rangle$) such that there exists a finite (respectively infinite) sequence of symbols $a_1, a_2, ..., a_n$ (respectively $a_1, a_2, ...$) called the label of ρ such that $\delta(q_i, a_i) = q_{i+1}$ for all $i \in \{0, ..., n-1\}$ (respectively $i \geq 0$). In fact, a path is a sequence of states that form a possible run of the automaton, and the label of this path is the input sequence that generates this run. The empty path is noted ϵ.

Let ρ be a path in some automaton, we write $inf(\rho)$ for the set of states that are visited infinitely often in ρ. The study of ω-automata was pioneered by Büchi in [7], where he introduced the Büchi automaton. The acceptance condition of such an automaton is given as a set of states, at least one of which must be visited infinitely often by a sequence for it to be accepted by the automaton. Formally:

Definition 1. *A Büchi automaton $\mathcal{B}$ is an ω-automaton $(Q, \mathcal{I}, \delta, F)$ whose acceptance condition is a set $F \subseteq Q$. A path ρ is valid iff $inf(\rho) \cap F \neq \emptyset$.*

A cycle of states from an automaton $\mathcal{A}$ is said to be a *valid cycle* iff the states composing it would respect the acceptance condition were they are the only states visited infinitely often in a path over this automaton. A cycle is said to be invalid otherwise.

Observe that in a non-deterministic Büchi automaton $\mathcal{B}$, only a single run of the automaton needs to be valid for its label to be a word accepted by $\mathcal{L}_{\mathcal{B}}$. The class of languages recognizable by Büchi automata is termed ω-regular languages [36]. A language $\mathcal{L} \subseteq \Sigma^{\omega}$ is said to be ω-regular iff it is of the form UV^{ω} where $U, V \subseteq \Sigma^{*}$ are regular languages. The use of nondeterministic automata sometimes adds a layer of complexity to proofs and automata manipulations. A deterministic automaton is thus sometimes preferable. Unfortunately, deterministic Büchi automata are strictly less expressive than their nondeterministic counterparts [36].

Yet, with altered acceptance conditions, an automaton class can be defined which is deterministic but still recognizes all ω-regular languages. Several such automata exist. In this paper, we focus on the Rabin automaton [37].

Definition 2. *A Rabin automaton $\mathcal{R}$ is an ω-automaton $(Q, \mathcal{I}, \delta, C)$ whose acceptance condition C is given as a set of pairs of sets of states (G_i, R_i) with $G_i, R_i \subseteq Q$. A run ρ is accepted iff there exists a pair $(G_i, R_i) \in C$ for which $inf(\rho) \cap G_i \neq \emptyset \ \wedge \ inf(\rho) \cap R_i = \emptyset$.*

4 Monitor Inlining Algorithm

In [11], Chabot et al. show how a safety or non-safety property can be inlined into an untrusted or possibly malicious program to produce an instrumented version of this program that provably respects the security policy, while maintaining the original program's transparency, (i.e. leaving valid executions present in the original program unaltered [38]).

The method consists of 7 steps.

Property Encoding

The desired security policy is abstracted by a Rabin automaton. This allows much wider expressivity than previous approaches which relied upon the security automaton [2], a subclass of the Büchi automaton limited to express safety properties, while retaining determinism. Any ω-regular property can be stated in this formalism [36].

Program Abstraction

The program is abstracted into a Labeled Transition System (LTS), a widely used formalism for representing programs. Transformations can also ensure that this representation is deterministic, without loss of expressivity.

Automata Product

The next step is to construct the automata product of the property and the program. This results in a new Rabin automaton ($\mathcal{R}^T$), with its own acceptance condition. Since this new automaton accepts the intersection of the language accepted by the original product and that accepted by the property automaton, it would form a natural basis from which to build the instrumented program. However, because the acceptance condition of the Rabin automaton is stated in terms of which states an execution can or cannot visit infinitely often, it is impossible for the monitor to detect at runtime if the current execution is valid or not. The remainder of the method consists of transformations aimed at removing invalid behaviors from the product automaton, while preserving valid ones.

Marking the Halt States

Since the property is enforced by halting the execution, it is necessary to identify all program points where the execution can safely be aborted without violating the security policy. These are indicated by adding a transition to a halt state from any state where the monitor can abort the execution.

Detecting Valid and Invalid Behaviors

The next phase consists in extracting, if possible, a labeled transition system from the product automaton, by pruning it of states and transitions containing invalid cycles w.r.t. its acceptance condition. This process first involves parsing the product automaton into its strongly connected components (*scc*) and listing the cycles present in each of them. It's at the step of cycle detection that the algorithm's complexity grows to $\mathcal{O}(2^n)$, for an automaton containing n states. Each cycle is then checked against the acceptance condition, and each *scc* is labeled according to whether it contains either only valid cycles, only invalid cycles, both types of cycles, or no cycles (the trivial *scc*).

Program transformations

The next step is to construct the quotient graph of the product automaton, in which each node represents a *scc*. The nodes of this graph are then visited in reverse topological ordering in order to determine, for each one, whether it should be kept intact, altered or removed. Every *scc* containing invalid cycles must be deleted, to ensure correction w.r.t. the desired security policy, but every *scc* containing valid cycles must be preserved, to ensure transparency. This approach thus fails in 3 cases: first, if the product automaton contains an *scc* which exhibits both valid and invalid cycles, second, if an *scc* containing valid cycles is reachable from an *scc* containing invalid cycles, and third, if an *scc* containing invalid cycles does not

possess any valid prefixes where the execution could be aborted without ruling out some valid executions.

Concretization

Whenever the previous step is successful, the resulting automaton is a Rabin automaton exhibiting only valid executions, and can thus be treated as an LTS and concretized into an executable program. The resulting program still exhibits all valid behaviors present in the original program, but it possesses no invalid behaviors. Since this program is built from the product automaton of a program of size n and a property of size m it's size is in the order of $\mathcal{O}(m \cdot n)$.

While the method sometimes fail to produce a suitable instrumented code, this never occurs if the desired property is a safety property which can be enforced using existing approaches. This approach is thus strictly more expressive, and the main limitations to its wider use is the algorithmic complexity discussed above. In the following section, we propose three strategies that make the problem more tractable, and reduces the size of the final instrumented program.

5 Proposed Optimizations

In this section, we propose three possible optimizations to the algorithm introduced above.

5.1 Optimization 1: Avoiding cycle enumeration using Büchi Automata

Since the most time-consuming part of the algorithm is the enumeration and evaluation of each cycle, a natural way to reduce the algorithmic complexity of the method is to avoid this task. One option is to state the property in the form of a Büchi automaton, rather than Rabin automaton, as the acceptance condition of the former is stated only in terms of reaching certain states infinitely often. For an algorithm based on Büchi automaton to be equally expressive as one based on the Rabin automaton, we are forced to consider non-deterministic Büchi automata.

The phases of program abstraction, automata product and marking of the halt states proceed in the exact same manner as was the case using the original method.

The detection of the valid and invalid behaviors present in the product automaton proceeds as follows. First, we must once again detect the strongly connected components in the product automaton $\mathcal{R}^T$. This can be performed in linear time using Tarjan's algorithm [40]. We then check each *scc* for the presence of valid and invalid cycles separately.

Since a run of the Büchi automaton is accepting iff it visits an accepting state infinitely often, the presence of a valid cycle in an *scc* can be determined in linear time, namely in $\mathcal{O}(n+e)$ for an automaton with n states and e edges, simply by checking for the presence of an accepting state in the *scc*.

To detect whether or not an *scc* contains any invalid cycles, we have to verify that this *scc* would still contain a cycle after deletion of all its valid states with their incident edges. This task is accomplished using a modified version of Tarjan's *scc* detection

algorithm [40], altered to ignore any edge incident to a valid state. An invalid cycle is present if a run of this algorithm detects a non-trivial *scc*. It is not necessary to enumerate all cycles, which is what lead to exponential algorithmic complexity of the algorithm based on the Rabin automaton. Instead, the simple detection of the presence of a cycle can be performed in linear time, since Tarjan's algorithm has a complexity in the order of $\mathcal{O}(n + e)$ for an automaton with n states and e edges.

The remainder of the algorithm, namely deleting the *scc*s containing invalid cycles, if doing so is allowed, and concretization, are performed in the same manner whether a Rabin or Büchi automaton is used. The complexity of the overall monitor inlining algorithm thus passes from $\mathcal{O}(2^n)$ to a much more tractable $\mathcal{O}(n + e)$, where n is the size of the product automaton and e is the number of transitions it contains. This optimization does however come at a cost. Since the Büchi automaton is non-deterministic, and a sequence is accepted if *any* of its possible runs over the automaton visits an accepting state infinitely often, any run containing invalid cycles may reflect a label that also generates valid runs over the same automaton. Whenever the product automaton contains some behavior that prevent the algorithm from pruning it of all invalid behaviors, such as, for instance, an *scc* containing both valid and invalid sequences, it may be possible that any run of the automaton reaching these problematic *scc*s corresponds to a sequence which also reaches both valid and invalid *scc*s for which the property is monitorable. The algorithm thus unnecessarily fails to output an instrumented program.

It follows that stating the desired property using a Büchi automaton results in a somewhat more conservative approximation than was the case with a Rabin automaton. However, in practice, we found it very difficult to construct examples of real properties and real programs that were monitorable using the original algorithm based upon the Rabin automaton, but not using the optimization described above. Furthermore, a relatively simple reachability analysis of the automaton, may allow us to enforce the same set of properties (for any given program) in both cases.

It is also important to note that the main contribution of [11] is to extend previous work by proposing a method capable of enforcing some non-safety properties, something that was not possible with previous techniques. Although the approach proposed in [11] is strictly more expressive than the optimization presented in this section, whenever the security policy is a safety property it can be stated as a deterministic automaton [23], and can thus be enforced using either methods. Both the method from [11] and the one presented in this section are therefore strictly more expressive than those proposed in previous works.

5.2 Optimization 2: Optimizing cycle detection using reducibility

If we wish to keep the Rabin automaton as the basis for stating the desired security policy, but still wish to avoid the exponential overhead incurred when enumerating the set of cycles, another interesting option is to draw upon the particular shape of the automaton being transformed to optimize the cycle detection phase.

Automata representing programs or properties often have a particular shape, termed reducibility. Reducible graphs were first described in [18]. Intuitively, a graph is reducible iff the graph can be reduced to a singleton by iteratively applying two graph

transformations, T1 and T2. T1 removes self loops while T2 collapses a node with a single predecessor into that predecessor.

Such graphs exhibit several properties that make them attractive to static analysis. Amongst them is the fact that they contain no loop with multiple entry points, and no adjacent loops. Each strongly connected component has a single entry point. Several static analyzes and optimizations can be performed more efficiently when manipulating reducible graphs. Amongst them is cycle detection, which can be performed in time linear to the number of backedges present in the graph.

Reducible graphs arise naturally in graphs that model the control flow of programs written in many commonly used programming languages. Nonetheless, several widely used programming languages, such as C, C++ Lisp or Pascal can produce irreducible control flow. When this occurs, several algorithms exist to transform an irreducible graph to an equivalent (in the sense that it accepts the same sequences) reducible graph. The most common algorithms are based on node splitting. This technique consist in duplicating certain states, and modifying the control frow, until the graph becomes reducible. Various heuristics have been proposed to minimize the amount of duplication. While in the worst case, an exponential blowup in the size of the model is unavoidable, in general the size of the equivalent reducible graph is quite reasonable (in the order of a 2.9% increase for some heuristics [20]). Furthermore, in [3], an algorithm is given which can produce an equivalent reducible graph with the minimal amount of duplication.

In our case, even though both the program abstraction and security property are stated by reducible automaton, the product automaton itself may not be reducible since the automaton product transformation does not preserve reducibility. However, our experience shows that in most cases, the product automaton can be seen as "quasi-reducible" in the sense that it can be made reducible with minimal overhead. This is because the property is generally much smaller than the target program, and involves only a small subset of security relevant instructions. We have found that transforming the product automaton to reducible form before proceeding with the cycle detection is an efficient way to reduce the runtime of transformation procedure. In our tests, we have used Janssen et al.'s [20] algorithm to transform the automaton to reducible form and Hetch's algorithm described in [1] to compute the set of cycles.

5.3 Other Optimizations

The size of the final instrumented code is in the order of $\mathcal{O}(n \cdot m)$ for an original program of size n and a property of size m. This increase in the size of the final program resulted from our objective that the runtime tests themselves incur no overhead. Still, reducing the size of the state space of the product automaton would further speedup the program transformation, and optimize the final code.

Unfortunately, producing a minimal ω-automaton is a complex problem [14]. In the case of the Büchi automata, for instance, it is PSPACE-hard. This makes it difficult to reduce the product automaton after it is created, and benefit from the smaller space state during the cycle detection and program transformation phases. However, if the program transformation is successful, the final automaton takes a particular form, called a weak automaton, which allows efficient minimization to be performed. In [30] Löding

shows that the acceptance condition for this class of ω-automata can be restated into an equivalent normal form, a procedure that can be done in time $\mathcal{O}(n)$ for an automaton with n states, and that once in normal form the state space minimization algorithms devised for deterministic finite automata can be applied to this automaton to produce an equivalent minimal automaton. The problem of minimizing a Deterministic Finite Automata is well studied and the most efficient algorithms run in time $\mathcal{O}(n \cdot \log n)$, to minimize an automaton with n states [19, 42, 43].

A final optimization we can consider is to minimize the number of pairs present in the acceptance condition of the Rabin automaton. Any improvement in this respect greatly speeds up the runtime of the algorithm since every cycle must be checked against each pair.

The minimal number of pairs needed in any deterministic Rabin automaton that accepts a given language is called the Rabin Index of that language [41, 22], and various algorithms exist that can perform this minimization. While there can be a substantial blowup in the size of the state-space when these transformations are performed (given as $\mathcal{O}(n^p \cdot 4^{2p^2})$ in [9], $\mathcal{O}(n \cdot 2^{p \log p})$ in [15] and $\mathcal{O}(n \cdot p^k)$ in [24], for an automaton with n states, p pairs and a Rabin Index of k)[3] the fact that a Rabin automaton representing a real property typically has only a few states at most may make this transformations worthwhile.

6 Empirical Results

Our preliminary empirical results are very encouraging. Representative results are shown in Figures 2 and 3. To generate these results, we ran two test sets, each of which consisted of applying both algorithms using two versions (Rabin and a Büchi) of the same property on the same program. This property accepts all executions containing only a finite non-empty number of a actions and such that finite executions end with a b action. The program was then extended by arbitrarily selecting any one of it's nodes, transitions or subgraphs, and replicating it using a multiplicative constant ranging from 1 to 200. We are then able to compare the performance of both algorithms when applying the same property to LTSs that grow in size, but still maintain some similarity.

Figure 2 highlights the potential gains of the Büchi automaton-based algorithm, as it shows the results of the enforcement of the same property as was used in the running example in [11] on an LTS built from a clique, with multiplicative constants 1-9. The Rabin automaton capturing this property is reproduced in Figure 1

We obtained similar results when enforcing a real security property on a real program. We tested the approach using an interesting real-life property with both a safety and a liveness component. The safety component is a mutual exclusion property stating that two processes are never enabled simultaneously. This is combined with a liveness requirement that both processes always be eventually enabled. This property was stated as a Müller automaton [32] and was subsequently translated into both a Rabin and a

[3] Strictly speaking, this refer to the transformation of a Rabin automaton into a special form called the Chain automaton, which is the critical step in reducing the Rabin automaton's acceptance condition. Once a Rabin automaton is stated in this form, the algorithm to reduce its acceptance condition is given in [9, 8].

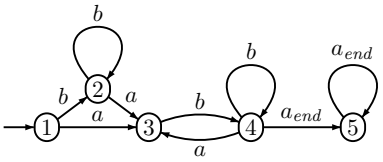

Fig. 1. The property from [11]

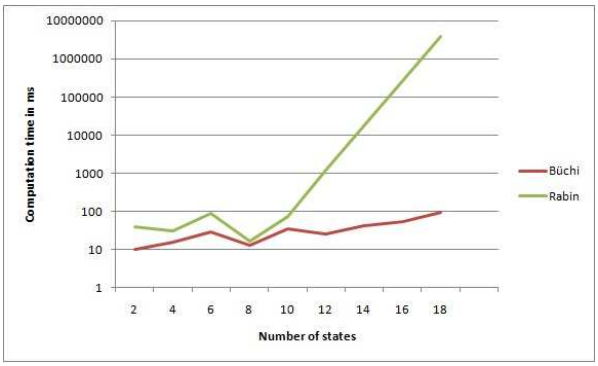

Fig. 2. Results of algorithms using the property from [11], with Rabin and Büchi.

Büchi automaton. We used a program whose control flow always respect the property. The program consists in multiple “transactions”, each of which represents a process that requests a resource, uses it, and then releases it. The program is extended by nesting multiple such transactions. The results, with multiplicative constants 1-200, can be seen in Figure 3.

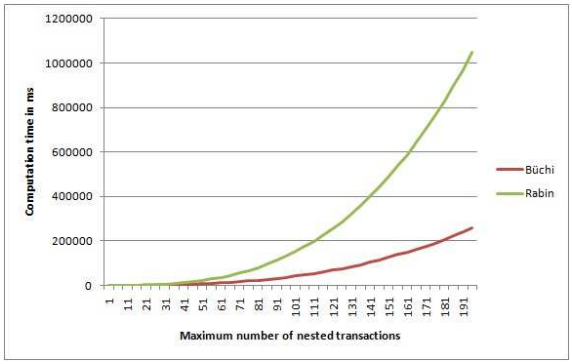

Fig. 3. Results of algorithms using the mutual exclusion property, with Rabin and Büchi.

Empirical results obtained while running the reducibility-based optimization proposed in section 5.2 are less striking, but still indicate that the optimization is worthwhile. However, because of the costs associated with the node-splitting transformation, this optimization is advantageous only if the synchronous product of the LTS and the property is both quasi-reducible and has a substantial size.

7 Conclusion and Future Work

In this paper, we propose several optimizations to an inline monitoring algorithm previously presented in the literature. These optimizations allow the inlining process to decrease the complexity from exponential to linear time in the size of the space-state of the product of the property being monitored and of the target program. We also show how the size of the resulting instrumented code can be minimized. These optimizations highlight a tradeoff between efficiency and the range of enforceable properties.

In future work, we hope to extend the algorithm so that it becomes possible to enforce the property in all cases. This may involve relying on monitors capable of more varied responses to potential violation of the security policy than simply aborting the execution.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

2. B. Alpern and F.B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:17–126, 1987.
3. Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Trans. Softw. Eng.*, 18:237–251, March 1992.
4. A. Bauer and J. Jürjens. Security protocols, properties, and their monitoring. In *Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems (SESS)*.
5. A. Bauer and J. Jürjens. Runtime verification of cryptographic protocols. *Computers & Security*, 29(3):315–330, 2010.
6. L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security*, Copenhagen, Denmark, July 2002.
7. J. Büchi. On a decision method in restricted second order arithmetic. In *In Proceedings of the International Congress on Logic, Method, and Philosophy of Science*, pages 1–12. Stanford University Press, Stanford, CA, 1962.
8. O. Carton. *Mots infinis, ω-semigroupes et topologie*. PhD thesis, Universite de Paris 07, 1993.
9. O. Carton. Chain automata. In *IFIP World Computer Congress '94*, pages 451–458, Hamburg, 1994. Elsevier (North-Holland).
10. H. Chabot, R. Khoury, and N. Tawbi. Generating in-line monitors for Rabin automata. In *Proceedings of the 14th Nordic Conference on Secure IT Systems, NordSec 2009*, volume 5838 of *LNCS*, pages 287–301. Springer, October 2009.
11. H. Chabot, R. Khoury, and N. Tawbi. Extending the enforcement power of truncation monitors using static analysis. *Computers & Security*, Forthcoming.
12. A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, pages 200–214.
13. T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Conference record of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2000.
14. R. Ehlers. Minimising deterministic Büchi automata precisely using sat solving. In *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, pages 326–332, 2010.
15. E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *SFCS '91: Proceedings of the 32nd annual symposium on Foundations of computer science*, pages 368–377, Washington, DC, USA, 1991. IEEE Computer Society.
16. U. Erlingsson and F.B. Schneider. Sasi enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop*. ACM Press, 2000.
17. G. Le Guernic. Automaton-based Confidentiality Monitoring of Concurrent Programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSFS20)*, pages 218–232. IEEE Computer Society, July 6–8 2007.
18. M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21:367–375, July 1974.
19. John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.
20. Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Programming Languages and Systems*, 19:1031–1052, 1997.
21. G. Morrisett K. W. Hamlen and F.B. Schneider. Computability classes for enforcement mechanisms. Technical Report TR2003-1908, Cornell University, 2003.
22. Michael Kaminski. A classification of omega-regular languages. *Theoretical Computer Science*, 36:217–229, 1985.
23. J. Klein. *Linear Time Logic and Deterministic omega-Automata*. PhD thesis, The University of Bonn, Bonn, Germany, January 2005.

24. S. C. Krishnan, A. Puri, R. K. Brayton, and P. P. Varaiya. The Rabin index and chain automata, with applications to automata and games. In *In Computer Aided Verification, Proc. 7th Int. Conference, LNCS 939*, pages 253–266, 1995.
25. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.
26. M. Langar and M. Mejri. Formal and efficient enforcement of security policies. In *Proceedings of The 2005 International Conference on Foundations of Computer Science, (FCS 2005)*, pages 143–149, 2005.
27. M. Langar, M. Mejri, and K. Adi. Formal monitor for concurrent programs. In *Workshop on Practice and Theory of IT Security*, 2006.
28. M. Langar, M. Mejri, and K. Adi. A formal approach for security policy enforcement in concurrent programs. In *Proceedings of the 2007 International Conference on Security & Management*, pages 165–171, 2007.
29. J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 12(3):1–41, 2009.
30. C. Löding. Efficient minimization of deterministic weak omega-automata. *Information Processing Letters*, 79:105–109(5), 31 July 2001.
31. T. Mechri, M. Langar, M. Mejri, H. Fujita, and Y. Funyu. Automatic enforcement of security in computer networks. In *New Trends in Software Methodologies, Tools and Techniques - Proceedings of the Sixth SoMeT 2007*, pages 200–222, 2007.
32. D. E. Muller. Infinite sequences and finite machines. *Switching Circuit Theory and Logical Design*, 0:3–16, 1963.
33. H. Ould-Slimane and M. Mejri. Enforcing security policies by rewriting programs using automata. In *Proceedings of the Workshop on Practice and Theory of IT Security (PTITS)*, pages 195–207, 2006.
34. H. Ould-Slimane, M. Mejri, and K. Adi. Enforcing security policies on programs. In *New Trends in Software Methodologies, Tools and Techniques - Proceedings of the Fifth SoMeT 2006, October 25-27, 2006, Quebec, Canada*, pages 195–207, 2006.
35. H. Ould-Slimane, M. Mejri, and K. Adi. Using edit automata for rewriting-based security enforcement. In *Data and Applications Security XXIII, 23rd Annual IFIP WG 11.3 Working Conference, Montreal, Canada, July 12-15, 2009. Proceedings*, pages 175–190, 2009.
36. D. Perrin and J. E. Pin. *Infinite Words*. Pure and Applied Mathematics Vol 141. Elsevier, 2004.
37. Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–37, 1969.
38. F.B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.
39. K. Sen, , A. Vardhan, G. Agha, and G. Roşu. Efficient decentralized monitoring of safety in distributed systems. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 418–427, Washington, DC, USA, 2004. IEEE Computer Society.
40. Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
41. Klaus Wagner. On omega-regular sets. *Information and Control*, 43(2):123–177, 1979.
42. Bruce W. Watson. A taxonomy of finite automata construction and minimization algorithms. Technical report, Computing Science, 1993.
43. Bruce W. Watson and Jan Daciuk. An efficient incremental dfa minimization algorithm. *Nat. Lang. Eng.*, 9(1):49–64, 2003.
44. F. Yan and P. W. L. Fong. Efficient irm enforcement of history-based access control policies. In *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2009, Sydney, Australia*, pages 35–46, 2009.