String Analysis via Automata Manipulation with Logic Circuit Representation

Hung-En Wang[1], Tzung-Lin Tsai[1], Chun-Han Lin[2], Fang Yu[2], and Jie-Hong R. Jiang[1(✉)]

[1] Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan
jhjiang@ntu.edu.tw

[2] Department of Management Information Systems, National Chengchi University, Taipei, Taiwan

Abstract. Many severe security vulnerabilities in web applications can be attributed to string manipulation mistakes, which can often be avoided through formal string analysis. String analysis tools are indispensable and under active development. Prior string analysis methods are primarily automata-based or satisfiability-based. The two approaches exhibit distinct strengths and weaknesses. Specifically, existing automata-based methods have difficulty in generating counterexamples at system inputs to witness vulnerability, whereas satisfiability-based methods are inadequate to produce filters amenable for firmware or hardware implementation for real-time screening of malicious inputs to a system under protection. In this paper, we propose a new string analysis method based on a scalable logic circuit representation for (nondeterministic) finite automata to support various string and automata manipulation operations. It enables both counterexample generation and filter synthesis in string constraint solving. By using the new data structure, automata with large state spaces and/or alphabet sizes can be efficiently represented. Empirical studies on a large set of open source web applications and well-known attack patterns demonstrate the unique benefits of our method compared to prior string analysis tools.

1 Introduction

Analyzing string manipulating code is of great importance because string manipulation is ubiquitous in modern software systems, such as web applications and database services. String analysis aims to determine the set of assignments to the string variables in string expressions that may arise from program execution or other sources. It can be applied, e.g., to identify security vulnerabilities by checking if a security sensitive function can receive an input string that contains an exploit [24,29,32], to identify behaviors of JavaScript code that use the `eval` function by computing the string values that can reach the `eval` function [15], to identify html generation errors by computing the html code generated by web applications [20], to identify the set of queries that are sent to back-end database

S. Chaudhuri and A. Farzan (Eds.): CAV 2016, Part I, LNCS 9779, pp. 241–260, 2016.
DOI: 10.1007/978-3-319-41528-4_13

by analyzing the code that generates the SQL queries [12], and to patch input validation and sanitization functions by automatically synthesizing repairs [31].

Prior string analysis methods are mainly automata-based or satisfiability-based. For automata-based approaches, explicit state-graph represented finite automata [8,13], MTBDD represented finite automata [2,32], and Boolean algebra represented symbolic finite automata [10,27,28] have been proposed. By characterizing a set of strings as a language, these methods are not restricted to particular bounds on string lengths. They can be used to synthesize filters or sanitizers [31] to screen out malicious string inputs to systems under protection, but have difficulty in generating counterexamples at system inputs to witness vulnerability. For satisfiability-based approaches, bit-vector based bounded checking [4,18,19,23] and satisfiability modulo theories (SMT) based constraint solving [1,3,26,34] have been proposed. They may answer a certain set of string queries with length constraints not doable for automata-based methods. By searching a solution to a given set of string constraints, they can generate counterexamples to witness vulnerability, but cannot support the synthesis of string filters amenable for firmware or hardware implementation for real-time screening of malicious inputs to a system under protection.

In this paper, we intend to support string analysis of acyclic constraints with both counterexample generation and filter synthesis capabilities. To achieve this goal, we develop a nondeterministic finite automata (NFA) manipulation engine with logic circuit representation. In particular, we adopt the and-inverter graph (AIG) [21], which have been widely adopted in logic synthesis for industrial applications in electronic design automation (EDA) in recent years, as the underlying data structure. Thereby automata manipulations can be performed implicitly using logic circuits while determinization is largely avoided. Our method is scalable to automata with large alphabet sizes in contrast to BDD-based automata representation. We further extend our method to represent symbolic finite automata [10], which may have infinite (or very large) alphabets [25]. Our method enables the generation of counterexamples for backtracking attack input strings to a vulnerable application and the synthesis of filters amenable for firmware or hardware implementation to avoid exploits of vulnerabilities in real time. The proposed method is implemented as a new string analysis tool, named SLOG. We conduct comprehensive experimental study on over 20000 string constraints generated from real web applications to compare state-of-the-art tools, including JSA [8], Stranger [30], Z3-str2 [34], CVC4 [3], and Norn [1]. Experiments suggest the performance advantage of SLOG in contrast to other string solvers with counterexample generation capabilities. Moreover, the scalability of SLOG is shown for automata with large alphabets in contrast to BDD-based methods of automata representation.

2 Preliminaries

A *finite automaton* A is a five-tuple (Q, Σ, I, T, O), where Q is a finite state set, Σ is an alphabet, $I \subseteq Q$ is a set of initial states, $T \subseteq \Sigma \times Q \times Q$ is a

state transition relation, and $O \subseteq Q$ is a set of accepting states. In the sequel, we shall instead represent the initial states, transition relation, and accepting states in terms of characteristic functions $I : Q \to \mathbb{B}$, $T : \Sigma \times Q \times Q \to \mathbb{B}$, and $O : Q \to \mathbb{B}$, respectively. (A characteristic function χ represents a (Boolean encoded) set S by having $\chi(e) = 1$ (TRUE) if $e \in S$ and $\chi(e) = 0$ (FALSE) if $e \notin S$.) A finite automaton can be either a deterministic finite automaton (DFA) or a nondeterministic finite automaton (NFA) depending on the determinicity of its state transition. In the sequel, we refer $\boldsymbol{x}$, $\boldsymbol{s}$ and $\boldsymbol{s}'$ to the input, current-state and next-state variables in the Boolean domain, and relate the valuations of variables $\boldsymbol{x}$, denoted $[\![\boldsymbol{x}]\!]$, and the valuations of variables $\boldsymbol{s}$, denoted $[\![\boldsymbol{s}]\!]$, to the alphabet Σ and state set Q, respectively. A *trace* of an automaton is a state-input alternating sequence $q_1, \sigma_1, q_2, \sigma_2, \ldots, q_\ell$, which satisfies $T(\sigma_i, q_i, q_{i+1})$ for $i = 1, \ldots, \ell - 1$.

A (finite) *string* $\sigma_1, \ldots, \sigma_n$, for $n \geq 0$ (an *empty string*, denoted ϵ, when $n = 0$), over alphabet Σ is accepted by an automaton if there exist states $q_1, \ldots, q_{n+1}$ such that $I(q_1) = 1$ (for q_1 being an initial state), $O(q_{n+1}) = 1$ (for q_{n+1} being an accepting state), and the sequence $q_1, \sigma_1, q_2, \sigma_2, \ldots, q_{n+1}$ forms a trace. The set of strings accepted by an automaton A is called the *(regular) language* accepted by A, denoted as $\mathcal{L}(A)$.

Because a finite automaton $A = (Q, \Sigma, I, T, O)$ can be fully described by the characteristic functions of I, T, and O, with Boolean encoding on Q and Σ the automaton A can be represented as a logic circuit, denoted $\mathcal{C}(A)$, that realizes these characteristic functions. In the sequel, we shall not distinguish between characteristic functions I, T, O and their circuit representations. In this work, we show how various string and automata manipulations can be achieved under the logic circuit representation of (nondeterministic) finite automata. For practical implementation, we exploit the *and-inverter graph (AIG)* [21] as the underlying data structure for scalable logic circuit representation and manipulation. An AIG is a directed acyclic graph $G(V, E)$, where each vertex $v \in V$ is either a primary input node without any fanin or a function node representing a two-input AND gate, and each edge $(u, v) \in E$ denotes a complemented or uncomplemented connection from vertex u to v. Due to its simplicity, the AIG has been efficiently implemented as a Boolean reasoning engine widely used in various logic synthesis and formal verification tasks in industrial very-large-scale integration (VLSI) designs.

In the sequel, we assume a finite automaton can be nondeterministic and may even take ϵ-transitions. To represent an ϵ-transition under the circuit representation, we reserve a symbol "ϵ" as an addendum to Σ with a special handling. Given a state transition relation T, we denote its equivalent variant with an ϵ self-transition inserted for each state as T^ϵ. That is, $T^\epsilon(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}')$ represents $T(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}') \vee ((\boldsymbol{s} = \boldsymbol{s}') \wedge (\boldsymbol{x} = \epsilon))$.

Given a web application and an attack pattern (specified as a regular expression) we can first extract *dependency graphs* for security sensitive functions, called the *sinks*, from the web application using static program analysis techniques [14,17]. Each extracted dependency graph shows how the input values

flow to a sink, including all the string operations performed on the input values before they reach the sink. A dependency graph is vulnerable if its sink node accepts an attack string (with respect to a given attack pattern). From the dependency graph, we can generate string constraint formulas and check whether the intersection of the sink node's language and the attack pattern is empty. If it is empty, then the web application is not vulnerable. Otherwise, a counterexample witnessing the vulnerability is to be computed.

3 String and Automata Operations

We show that string/language operations, including *intersection*, *union*, *concatenation*, *deletion*, *replacement*, and *emptiness checking*, can be achieved under logic circuit representation. We omit other less used operations, including reversion, prefix, suffix, and substring, due to space limitation.

In the following exposition we assume an automaton A (or A_i) is represented as a circuit of its characteristic functions $T(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}')$, $I(\boldsymbol{s})$, and $O(\boldsymbol{s})$ (or $T_i(\boldsymbol{x}, \boldsymbol{s}_i, \boldsymbol{s}'_i)$, $I_i(\boldsymbol{s}_i)$, and $O_i(\boldsymbol{s}_i)$ for $i = 1, 2, 3$). Also we assume without loss of generality that $|\boldsymbol{s}_1| = m$ and $|\boldsymbol{s}_2| = n$ for automata A_1 and A_2, respectively, with $m \leq n$ in our following discussion unless otherwise said.

3.1 Intersection

Given two automata A_1 and A_2, the automaton $A_{\text{INT}} = \text{INT}(A_1, A_2)$ that accepts language $\mathcal{L}(A_{\text{INT}}) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ is the product machine with the characteristic functions T_{INT}, I_{INT}, O_{INT} constructed by first augmenting the transition relations T_1 and T_2 to T_1^ϵ and T_2^ϵ, respectively, by inserting an ϵ self-transition for each state, and second conjuncting the resultant characteristic functions of A_1 and A_2. Accordingly, we have

$$\frac{(T_1, I_1, O_1) \qquad (T_2, I_2, O_2)}{(T_{\text{INT}}, I_{\text{INT}}, O_{\text{INT}})}\ \text{INT}$$

with

$$\begin{aligned} T_{\text{INT}}(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}') &= T_1^\epsilon(\boldsymbol{x}, \boldsymbol{s}_1, \boldsymbol{s}'_1) \wedge T_2^\epsilon(\boldsymbol{x}, \boldsymbol{s}_2, \boldsymbol{s}'_2), \\ I_{\text{INT}}(\boldsymbol{s}) &= I_1(\boldsymbol{s}_1) \wedge I_2(\boldsymbol{s}_2), \\ O_{\text{INT}}(\boldsymbol{s}) &= O_1(\boldsymbol{s}_1) \wedge O_2(\boldsymbol{s}_2), \end{aligned}$$

for $\boldsymbol{s} = (\boldsymbol{s}_1, \boldsymbol{s}_2)$. The corresponding circuit construction is illustrated in Fig. 1(a). The constructed circuit is of size $O(|\mathcal{C}(A_1)| + |\mathcal{C}(A_2)|)$ and has $(|\boldsymbol{s}_1| + |\boldsymbol{s}_2|)$ state variables.

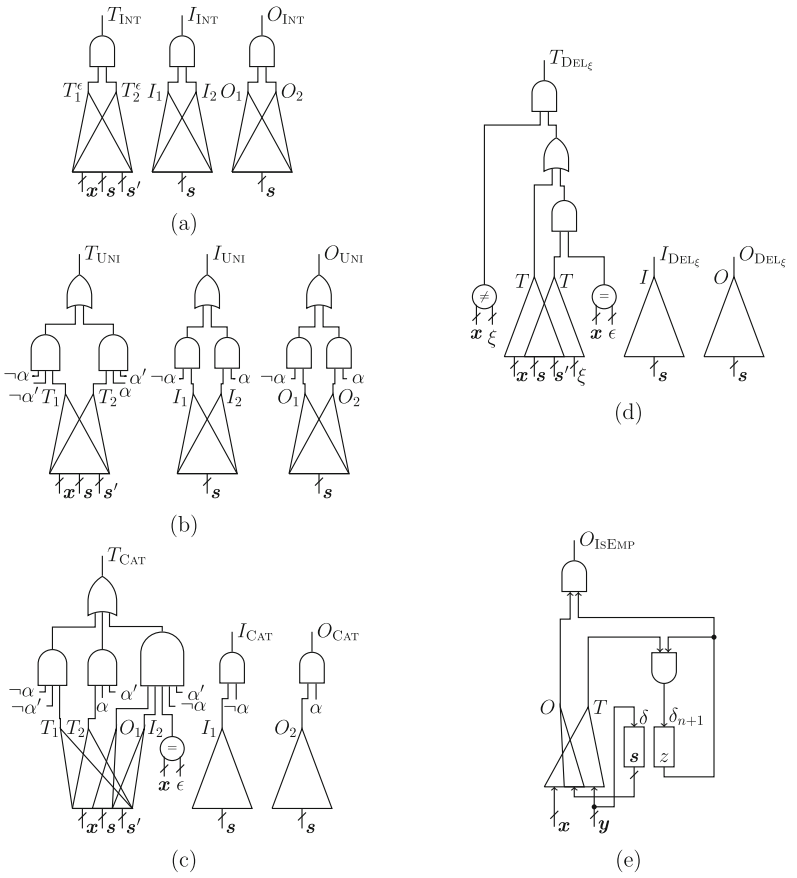

Fig. 1. Circuit construction of (a) INT, (b) UNI, (c) CAT, (d) DEL_ξ, and (e) ISEMP operations.

3.2 Union

Given two automata A_1 and A_2, the automaton $A_{\text{UNI}} = \text{UNI}(A_1, A_2)$ that accepts language $\mathcal{L}(A_{\text{UNI}}) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$ can be constructed by disjointly unioning the two with state variables being merged and states being distinguished by an auxiliary variable α, similar to the multiplexed machine in [16], as follows.

$$\frac{(T_1, I_1, O_1) \qquad (T_2, I_2, O_2)}{(T_{\text{UNI}}, I_{\text{UNI}}, O_{\text{UNI}})}\ \text{UNI}$$

with

$$\begin{aligned} T_{\text{UNI}}(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}') &= (\neg\alpha \wedge \neg\alpha' \wedge T_1(\boldsymbol{x}, \langle \boldsymbol{s}_2 \rangle_m, \langle \boldsymbol{s}_2' \rangle_m)) \vee (\alpha \wedge \alpha' \wedge T_2(\boldsymbol{x}, \boldsymbol{s}_2, \boldsymbol{s}_2')), \\ I_{\text{UNI}}(\boldsymbol{s}) &= (\neg\alpha \wedge I_1(\langle \boldsymbol{s}_2 \rangle_m)) \vee (\alpha \wedge I_2(\boldsymbol{s}_2)), \\ O_{\text{UNI}}(\boldsymbol{s}) &= (\neg\alpha \wedge O_1(\langle \boldsymbol{s}_2 \rangle_m)) \vee (\alpha \wedge O_2(\boldsymbol{s}_2)), \end{aligned}$$

where $\boldsymbol{s} = (\boldsymbol{s}_2, \alpha)$ and the bracket "$\langle \boldsymbol{s}_2 \rangle_m$" indicates taking a subset of the first m variables of $\boldsymbol{s}_2$. Essentially the state variables $\boldsymbol{s}$ of A_1 is merged into $\boldsymbol{s}_2$ so that the first m variables of $\boldsymbol{s}_2$ are shared by both A_1 and A_2. Moreover, the α bit signifies the states of A_1 by $\alpha = 0$, and signifies the states of A_2 by $\alpha = 1$. That is, a state $q \in [\![\boldsymbol{s}]\!]$ belongs to A_1 if its variable α valuates to 0, and to A_2 if α valuates to 1. The corresponding circuit construction is illustrated in Fig. 1(b). The constructed circuit is of size $O(|\mathcal{C}(A_1)| + |\mathcal{C}(A_2)|)$ and has $(\max\{|\boldsymbol{s}_1|, |\boldsymbol{s}_2|\} + 1)$ state variables.

3.3 Concatenation

Given two automata A_1 and A_2, the automaton $A_{\text{CAT}} = \text{CAT}(A_1, A_2)$ that accepts language $\mathcal{L}(A_{\text{CAT}}) = (\mathcal{L}(A_1).\mathcal{L}(A_2))$, which contains the set of concatenated strings $\boldsymbol{\sigma}_1.\boldsymbol{\sigma}_2$ for $\boldsymbol{\sigma}_1 \in \mathcal{L}(A_1)$ and $\boldsymbol{\sigma}_2 \in \mathcal{L}(A_2)$, can be constructed, in a way similar to UNI, as follows.

$$\frac{(T_1, I_1, O_1) \qquad (T_2, I_2, O_2)}{(T_{\text{CAT}}, I_{\text{CAT}}, O_{\text{CAT}})}\ \text{CAT}$$

with

$$\begin{aligned} T_{\text{CAT}}(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}') = {} & (\neg\alpha \wedge \neg\alpha' \wedge T_1(\boldsymbol{x}, \langle \boldsymbol{s}_2 \rangle_m, \langle \boldsymbol{s}_2' \rangle_m)) \vee (\alpha \wedge \alpha' \wedge T_2(\boldsymbol{x}, \boldsymbol{s}_2, \boldsymbol{s}_2')) \vee \\ & ((\boldsymbol{x} = \epsilon) \wedge \neg\alpha \wedge \alpha' \wedge O_1(\langle \boldsymbol{s}_2 \rangle_m) \wedge I_2(\boldsymbol{s}_2')), \\ I_{\text{CAT}}(\boldsymbol{s}) = {} & \neg\alpha \wedge I_1(\langle \boldsymbol{s}_2 \rangle_m), \\ O_{\text{CAT}}(\boldsymbol{s}) = {} & \alpha \wedge O_2(\boldsymbol{s}_2), \end{aligned}$$

for $\boldsymbol{s} = (\boldsymbol{s}_2, \alpha)$. The corresponding circuit construction is illustrated in Fig. 1(c). The constructed circuit is of size $O(|\mathcal{C}(A_1)| + |\mathcal{C}(A_2)|)$ and has $(\max\{|\boldsymbol{s}_1|, |\boldsymbol{s}_2|\} + 1)$ state variables.

3.4 Deletion

Given an automaton A, the automaton $A_{\text{DEL}_\xi} = \text{DEL}(A, \xi)$ for $\xi \in \Sigma$ that accepts the strings of $\boldsymbol{\sigma} \in \mathcal{L}(A)$ but with each appearance of symbol "ξ" in $\boldsymbol{\sigma}$ being removed can be constructed as follows.

$$\frac{(T, I, O)}{(T_{\text{DEL}_\xi}, I_{\text{DEL}_\xi}, O_{\text{DEL}_\xi})}\ \text{DEL}_\xi$$

with

$$\begin{aligned} T_{\text{DEL}_\xi}(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}') &= (T(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}') \vee ((\boldsymbol{x} = \epsilon) \wedge T(\xi, \boldsymbol{s}, \boldsymbol{s}'))) \wedge (\boldsymbol{x} \neq \xi), \\ I_{\text{DEL}_\xi}(\boldsymbol{s}) &= I(\boldsymbol{s}), \\ O_{\text{DEL}_\xi}(\boldsymbol{s}) &= O(\boldsymbol{s}), \end{aligned}$$

(The deletion operation is a special case of the replacement operation by replacing an alphabet symbol with ϵ.) The corresponding circuit construction is illustrated in Fig. 1(d). The constructed circuit is of size $O(|\mathcal{C}(A)|)$ and has $|\boldsymbol{s}|$ state variables.

3.5 Replacement

Given three automata A_1, A_2, A_3, we study how to construct the automata $A_{\text{REP}} = \text{REP}(A_1, A_2, A_3)$ that accepts the language $\{(\boldsymbol{\sigma_1}.\boldsymbol{\tau}_1.\boldsymbol{\sigma}_2.\boldsymbol{\tau}_2 \ldots) \in \Sigma^* \mid (\boldsymbol{\sigma_1}.\boldsymbol{\rho}_1.\boldsymbol{\sigma}_2.\boldsymbol{\rho}_2 \ldots) \in \mathcal{L}(A_1),\ \boldsymbol{\sigma_i} \notin (\Sigma^*.\mathcal{L}(A_2).\Sigma^*),\ \boldsymbol{\rho}_i \in \mathcal{L}(A_2)$ and $\boldsymbol{\tau}_i \in \mathcal{L}(A_3)$ for all $i\}$, that is, replacing $\mathcal{L}(A_2)$ with $\mathcal{L}(A_3)$ in $\mathcal{L}(A_1)$. Based upon [32], we construct the automata A_{REP} as follows.

$$\frac{(T_1, I_1, O_1) \qquad (T_2, I_2, O_2) \qquad (T_3, I_3, O_3)}{(T_{\text{REP}}, I_{\text{REP}}, O_{\text{REP}})}\ \text{REP}$$

First, we build automaton $A_1^{\lhd\rhd}$, which parenthesizes any substrings of a string in $\mathcal{L}(A_1)$ by two fresh new symbols "$\lhd$" and "$\rhd$". It yields from A_1 the automaton $A_1^{\lhd\rhd}$ with

$$\begin{aligned}
T_1^{\lhd\rhd} &= ((\alpha = \alpha') \wedge (\boldsymbol{x} \neq \lhd) \wedge (\boldsymbol{x} \neq \rhd) \wedge T_1(\boldsymbol{x}, \boldsymbol{s}_1, \boldsymbol{s}_1')) \vee \\
&\quad ((\boldsymbol{s}_1 = \boldsymbol{s}_1') \wedge ((\neg\alpha \wedge \alpha' \wedge (\boldsymbol{x} = \lhd)) \vee (\alpha \wedge \neg\alpha' \wedge (\boldsymbol{x} = \rhd)))), \\
I_1^{\lhd\rhd} &= \neg\alpha \wedge I_1(\boldsymbol{s}_1), \\
O_1^{\lhd\rhd} &= \neg\alpha \wedge O_1(\boldsymbol{s}_1).
\end{aligned}$$

The above construction makes two copies of the state space distinguished by variable α. When the input symbol is not equal to $\lhd$ or $\rhd$, the state transition is the same as A_1. When the input symbol equals $\lhd$ (resp. $\lhd$), the state in the $\alpha = 0$ (resp. $\alpha = 1$) space transitions to its counterpart in the $\alpha = 1$ (resp. $\alpha = 0$) space.

Second, we build automaton A_4, which is the automaton that accepts the strings $\{(\boldsymbol{\sigma}_1. \lhd .\boldsymbol{\rho}_1. \rhd .\boldsymbol{\sigma}_2. \lhd .\boldsymbol{\rho}_2. \rhd \ldots) \in \Sigma^* \mid \boldsymbol{\sigma}_i \in \overline{\Sigma^*.\mathcal{L}(A_2).\Sigma^*}$ and $\boldsymbol{\rho}_i \in \mathcal{L}(A_2)\}$. Let A_h be the automaton that accepts the language $\overline{\Sigma^*.\mathcal{L}(A_2).\Sigma^*}$ with characteristic functions $T_h(\boldsymbol{x}, \boldsymbol{s}_h, \boldsymbol{s}_h')$, $I_h(\boldsymbol{s}_h)$, $O_h(\boldsymbol{s}_h)$. Notice that constructing the automaton A_h requires complementing an NFA and is of exponential cost. Fortunately in most applications the automaton A_2 is known *a priori* and thus can be precomputed. Given A_2 and A_h, assuming without loss of generality $|\boldsymbol{s}_h| = n \geq |\boldsymbol{s}_2| = m$, automata A_4 can be derived as follows.

$$\begin{aligned}
T_4 &= (\neg\beta \wedge \neg\beta' \wedge (\boldsymbol{x} \neq \lhd) \wedge (\boldsymbol{x} \neq \rhd) \wedge T_h(\boldsymbol{x}, \boldsymbol{s}_h, \boldsymbol{s}_h')) \vee \\
&\quad (\beta \wedge \beta' \wedge (\boldsymbol{x} \neq \lhd) \wedge (\boldsymbol{x} \neq \rhd) \wedge T_2(\boldsymbol{x}, \langle \boldsymbol{s}_h \rangle_m, \langle \boldsymbol{s}_h' \rangle_m)) \vee \\
&\quad (\neg\beta \wedge \beta' \wedge (\boldsymbol{x} = \lhd) \wedge O_h(\boldsymbol{s}_h) \wedge I_2(\langle \boldsymbol{s}_h' \rangle_m)) \vee \\
&\quad (\beta \wedge \neg\beta' \wedge (\boldsymbol{x} = \rhd) \wedge O_2(\langle \boldsymbol{s}_h \rangle_m) \wedge I_h(\boldsymbol{s}_h')), \\
I_4 &= \neg\beta \wedge I_h(\boldsymbol{s}_h), \\
O_4 &= \neg\beta \wedge (O_h(\boldsymbol{s}_h) \vee I_h(\boldsymbol{s}_h)).
\end{aligned}$$

Third, let $A_5 = \text{INT}(A_1^{\lhd\rhd}, A_4)$ with characteristic functions $T_5(\boldsymbol{x}, \boldsymbol{s}_5, \boldsymbol{s}_5')$, $I_5(\boldsymbol{s}_5)$, $O_5(\boldsymbol{s}_5)$, where $\boldsymbol{s}_5 = (\boldsymbol{s}_1, \alpha, \boldsymbol{s}_4)$ with $\boldsymbol{s}_4 = (\boldsymbol{s}_h, \beta)$. Hence A_5 accepts the strings in $\mathcal{L}(A_1)$ with all the substrings in $\mathcal{L}(A_2)$ being marked. Then, in

$\mathcal{L}(A_5)$ instead of replacing substrings $\lhd\mathcal{L}(A_2)\rhd$ with strings in $\mathcal{L}(A_3)$, we replace $\lhd$ with $\mathcal{L}(A_3)$, $\rhd$ with ϵ, and $\mathcal{L}(A_2)$ with ϵ. We obtain

$$\begin{aligned}
T_{\text{REP}}(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}') = {} & (\neg\alpha \wedge \neg\alpha' \wedge T_5(\boldsymbol{x}, \boldsymbol{s}_5, \boldsymbol{s}'_5) \wedge \neg\gamma \wedge \neg\gamma' \wedge I_3(\boldsymbol{s}_3) \wedge I_3(\boldsymbol{s}'_3)) \vee \\
& (\neg\alpha \wedge \neg\alpha' \wedge (\boldsymbol{s}_5 = \boldsymbol{s}'_5) \wedge (\boldsymbol{x} = \epsilon) \wedge \neg\gamma \wedge \gamma' \wedge I_3(\boldsymbol{s}_3) \wedge I_3(\boldsymbol{s}'_3)) \vee \\
& (\neg\alpha \wedge \neg\alpha' \wedge (\boldsymbol{s}_5 = \boldsymbol{s}'_5) \wedge \gamma \wedge \gamma' \wedge T_3(\boldsymbol{x}, \boldsymbol{s}_3, \boldsymbol{s}'_3)) \vee \\
& (\neg\alpha \wedge \alpha' \wedge T_5(\lhd, \boldsymbol{s}_5, \boldsymbol{s}'_5) \wedge (\boldsymbol{x} = \epsilon) \wedge \gamma \wedge \neg\gamma' \wedge I_3(\boldsymbol{s}'_3) \wedge O_3(\boldsymbol{s}_3)) \vee \\
& (\alpha \wedge \alpha' \wedge \exists \boldsymbol{y}.[T_5(\boldsymbol{y}, \boldsymbol{s}_5, \boldsymbol{s}'_5)] \wedge (\boldsymbol{x} = \epsilon) \wedge \neg\gamma \wedge \neg\gamma' \wedge I_3(\boldsymbol{s}_3) \wedge I_3(\boldsymbol{s}'_3)) \vee \\
& (\alpha \wedge \neg\alpha' \wedge T_5(\rhd, \boldsymbol{s}_5, \boldsymbol{s}'_5) \wedge (\boldsymbol{x} = \epsilon) \wedge \neg\gamma \wedge \neg\gamma' \wedge I_3(\boldsymbol{s}_3) \wedge I_3(\boldsymbol{s}'_3)), \\
I_{\text{REP}}(\boldsymbol{s}) = {} & \neg\gamma \wedge I_5(\boldsymbol{s}_5) \wedge I_3(\boldsymbol{s}_3), \\
O_{\text{REP}}(\boldsymbol{s}) = {} & \neg\gamma \wedge O_5(\boldsymbol{s}_5) \wedge I_3(\boldsymbol{s}_3),
\end{aligned}$$

for $\boldsymbol{s} = (\boldsymbol{s}_5, \boldsymbol{s}_3, \gamma)$.

The constructed circuit is of size $O(|\mathcal{C}(A_1)| + |\mathcal{C}(A_2)| + |\mathcal{C}(A_h)| + |\mathcal{C}(A_3)|)$ and has $|\boldsymbol{x}|$ quantified internal variables.

3.6 Emptiness Checking

One important query, $\text{IsEmp}(A)$, about an automaton A is asking whether the language $\mathcal{L}(A)$ is empty. We employ property directed reachability (PDR) [11], an implementation of the state-of-the-art model checking algorithm IC3 [5] in the Berkeley ABC system [6], to test whether an accepting state is reachable from an initial state in A. Note that PDR only accepts a sequential circuit specified in transition functions, rather than a transition relation, as input; furthermore, it assumes the given circuit shall have a single initial state. Unfortunately because our automata are nondeterministic in general, their nondeterministic transitions can only be specified using transition relations and they may have multiple initial states.

To overcome the above mismatch between transition relation and transition function, we devise a mechanism converting $(T(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}'), I(\boldsymbol{s}), O(\boldsymbol{s}))$ representation of NFA A into a form acceptable by PDR as follows. To handle the single initial state restriction, let A_ϵ be the automaton accepting only the ϵ string, which is composed of a single initial accepting state without any transition. We modify A by $\text{Cat}(A_\epsilon, A)$ to enforce a single initial state. Moreover, to convert a transition relation to a set of transition functions, we introduce n new input variables $\boldsymbol{y}$ for $n = |\boldsymbol{s}|$ and a new state variable z with initial value 1, and construct a new sequential circuit with

- the output function: $O_{\text{IsEmp}} = (O(\boldsymbol{s}) \wedge z)$, and
- the next-state functions: $\delta_i = (y_i)$ for state variables s_i, $i = 1, \ldots, n$, and $\delta_{n+1} = (T(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{y}) \wedge z)$ for the state variable z.

Fig. 1(e) shows the corresponding circuit construction, where the rectangular boxes denote state-holding elements. With these conversions, PDR can be directly applied on the constructed new circuit. The constructed circuit is of size $O(\mathcal{C}(A))$ and has $(|\boldsymbol{s}| + 1)$ state variables and $(|\boldsymbol{x}| + |\boldsymbol{y}|)$ input variables. The complexity of checking language emptiness is PSPACE-complete in the circuit size of the underlying automaton.

4 Counterexample Generation

The automata manipulation flow specified in a dependency graph often ends with an IsEmp query asking whether a vulnerability exists for the application under verification. If the answer to IsEmp is negative, it is desirable to generate a counterexample witnessing the vulnerability. Such a counterexample should be expressed in terms of the inputs to the application. However since the counterexample to the IsEmp query is a trace demonstrating the reachability from an initial state to an accepting state in the final automaton, it does not directly correspond to the counterexample at the inputs. By counterexample generation, we compute counterexample traces at the inputs of a dependency graph that together induce a specific counterexample trace at the sink node. Prior automata-based methods cannot easily generate such counterexamples because the output automaton resulted from an automata operation does not contain information about its input automata whereas our circuit construction preserves such information through the introduced auxiliary variables.

Below we show how to backtrack from the negative answer to IsEmp to extract the input counterexamples. The backtrack process traverses the dependency graph in a reverse topological order and deduces the upstream counterexamples according to the corresponding operations in the following. Notice that an automata circuit iteratively constructed by our method may contain internally quantified variables. These variables are treated as free variables in PDR computation without explicit quantifier elimination, and their corresponding assignments are determined by PDR and returned along with the trace information.

Intersection. Let $(p_1, q_1), (\sigma_1, \rho_1, \varrho_1), (p_2, q_2), (\sigma_2, \rho_2, \varrho_2), \ldots, (p_\ell, q_\ell)$ be the counterexample trace of automaton $A_{\text{Int}} = \text{Int}(A_1, A_2)$, where $p_i \in [\![\boldsymbol{s}_1]\!], q_i \in [\![\boldsymbol{s}_2]\!], \sigma_i \in \Sigma$, $\rho_i \in \Sigma^k$, $\varrho_i \in \Sigma^l$, for some $k, l \geq 0$ and $(\boldsymbol{s}_1, \boldsymbol{s}_2)$ being the state variables of A_{Int} as constructed in Sect. 3.1. Let the values $\rho_i \in \Sigma^k$ and $\varrho_i \in \Sigma^l$ correspond to the assignments to the internally quantified variables of A_1 and A_2, respectively. Then the counterexample traces of A_1 and A_2 can be extracted backward by the following rule.

$$\frac{A_1\colon (p_1, (\sigma_1, \rho_1), \ldots, p_\ell) \qquad A_2\colon (q_1, (\sigma_1, \varrho_1), \ldots, q_\ell)}{A_{\text{Int}}\colon ((p_1, q_1), (\sigma_1, \rho_1, \varrho_1), \ldots, (p_\ell, q_\ell))}\ \textsc{IntCex}$$

Union. Let $(q_1, c), (\sigma_1, \rho_1), (q_2, c), (\sigma_2, \rho_2), \ldots, (q_\ell, c)$ be the counterexample trace of automaton $A_{\text{Uni}} = \text{Uni}(A_1, A_2)$, where $q_i \in [\![\boldsymbol{s}_2]\!]$, $c \in [\![\alpha]\!]$, $\sigma_i \in \Sigma$, and $\rho_i \in \Sigma^k$, for some $k \geq 0$ and $\boldsymbol{s}_2$ being the state variables of A_{Uni} as constructed in Sect. 3.2. Let the values $\rho_i \in \Sigma^k$ correspond to the assignments to the internally quantified variables of A_1 or A_2. The the counterexample traces of A_1 and A_2 can be extracted backward by the following rules.

$$\frac{A_1\colon (q_1, (\sigma_1, \rho_1), \ldots, q_\ell) \qquad A_2\colon (\bot)}{A_{\text{UNI}}\colon ((q_1, c), (\sigma_1, \rho_1), \ldots, (q_\ell, c))}\ \textsc{UniCex},\ c = 0$$

$$\frac{A_1\colon (\bot) \qquad A_2\colon (q_1, (\sigma_1, \rho_1), \ldots, q_\ell)}{A_{\text{UNI}}\colon ((q_1, c), (\sigma_1, \rho_1), \ldots, (q_\ell, c))}\ \textsc{UniCex},\ c = 1$$

Concatenation. Let (q_1, c_1), (σ_1, ρ_1), (q_2, c_2), (σ_2, ρ_2), $\ldots$, (q_ℓ, c_ℓ) be the counterexample trace of automaton $A_{\text{CAT}} = \textsc{Cat}(A_1, A_2)$, where $q_i \in [\![\boldsymbol{s}_2]\!]$, $c_i \in [\![\alpha]\!]$, $\sigma_i \in \Sigma$, and $\rho_i \in \Sigma^{k_i}$, for some $k_i \geq 0$ and $(\boldsymbol{s}_2, \alpha)$ being the state variables of A_{CAT} as constructed in Sect. 3.3. Let the values $\rho_i \in \Sigma^{k_i}$ correspond to the assignments to the internally quantified variables of A_1 or A_2. Then the counterexample traces of A_1 and A_2 can be extracted backward by the following rule.

$$\frac{A_1\colon (q_1, z_1, \ldots, q_i) \qquad A_2\colon (q_{i+1}, z_{i+1}, \ldots, q_n)}{A_{\text{CAT}}\colon (p_1, z_1 \ldots, p_i, z_i, p_{i+1}, z_{i+1}, \ldots, p_\ell)}\ \textsc{CatCex}$$

where each $p_j = (q_j, 0)$ for all $j \leq i$, $p_j = (q_j, 1)$ for all $j \geq i{+}1$, and $z_j = (\sigma_j, \rho_j)$ for all $j \neq i$, and $z_i = (\epsilon, \rho_i)$.

Replacement. Let $(p_1,\ c_1,\ q_1,\ r_1,\ d_1)$, $(\sigma_1,\ \rho_1,\ \varrho_1)$, $\ldots$, $(p_{n_1},\ c_{n_1},\ q_{n_1},\ r_{n_1},\ d_{n_1})$, $(\sigma_{n_1},\ \rho_{n_1},\ \varrho_{n_1})$, $(p_{n_1+1},\ c_{n_1+1},\ q_{n_1+1},\ r_{n_1+1},\ d_{n_1+1})$, $(\sigma_{n_1+1},\ \rho_{n_1+1},\ \varrho_{n_1+1})$, $\ldots$, $(p_{n_2},\ c_{n_2},\ q_{n_2},\ r_{n_2},\ d_{n_2})$, $(\sigma_{n_2},\ \rho_{n_2},\ \varrho_{n_2})$, $(p_{n_2+1},\ c_{n_2+1},\ q_{n_2+1},\ r_{n_2+1},\ d_{n_2+1})$, $(\sigma_{n_2+1},\ \rho_{n_2+1},\ \varrho_{n_2+1})$, $\ldots$, $(p_\ell,\ c_\ell,\ q_\ell,\ r_\ell,\ d_\ell)$ be the counterexample trace of automaton $A_{\text{REP}} = \textsc{Rep}(A_1, A_2, A_3)$, where $p_i \in [\![\boldsymbol{s}_1]\!]$, $c_i \in [\![\alpha]\!]$, $q_i \in [\![\boldsymbol{s}_4]\!]$, $r_i \in [\![\boldsymbol{s}_3]\!]$, $d_i \in [\![\gamma]\!]$, $\sigma_i, \varrho_i \in \Sigma$, and $\rho_i \in \Sigma^k$, for some $k \geq 0$ and $(\boldsymbol{s}_1, \alpha, \boldsymbol{s}_4, \boldsymbol{s}_3, \gamma)$ being the state variables of A_{REP} as constructed in Sect. 3.5. The trace must have the following form: Consider $(p_{n_i+1}, c_{n_i+1}, q_{n_i+1}, r_{n_i+1}, d_{n_i+1}), (\sigma_{n_i+1}, \rho_{n_i+1}, \varrho_{n_i+1}), \ldots, (p_{n_{i+1}}, c_{n_{i+1}}, q_{n_{i+1}}, r_{n_{i+1}}, d_{n_{i+1}})$. (Notice the subtle subscript difference between $n_i + 1$ and n_{i+1}.) For $i = 3m$, we have $c_j = 0$, $d_j = 0$ for $n_i + 1 \leq j \leq n_{i+1}$, and $\sigma_{n_i+1}\sigma_{n_i+2} \ldots \sigma_{n_{i+1}-1} \notin \Sigma^*.\mathcal{L}(A_2).\Sigma^*$. For $i = 3m{+}1$, we have $c_j = 0, d_j = 1$ for $n_i + 1 \leq j \leq n_{i+1}$, and $\sigma_{n_i+1}\sigma_{n_i+2} \ldots \sigma_{n_{i+1}-1} \in \mathcal{L}(A_3)$. For $i = 3m + 2$, we have $c_j = 1$, $d_j = 0$, $\sigma_j = \epsilon$ for $n_i + 1 \leq j \leq n_{i+1}$, and $\varrho_{n_i+1}\varrho_{n_i+2} \ldots \varrho_{n_{i+1}-1} \in \mathcal{L}(A_2)$. Also $\sigma_{n_i} = \epsilon$ for all i.

Let the values $\rho_i \in \Sigma^k$ and $\varrho_i \in \Sigma$ correspond to the assignments to the internally quantified variables of A_1 and to the assignments to the internally quantified variables added in the construction of A_{REP}, respectively. Then the counterexample trace of A_1 can be extracted backward by the following rule.

$$\frac{A_1\colon ((\omega_1^\dagger)^-, (\omega_3^\ddagger)^-, (\omega_4^\dagger)^-, (\omega_6^\ddagger)^-, \ldots, (\omega_\ell^\dagger))}{A_{\text{REP}}\colon (\omega_1, z_1, \omega_2, z_2, \omega_3, z_3, \omega_4, z_4, \omega_5, z_5, \omega_6, z_6, \ldots, \omega_\ell)}\ \textsc{RepCex}$$

where each ω_i denote the trace $(p_{n_{i-1}+1},\ c_{n_{i-1}+1},\ q_{n_{i-1}+1},\ r_{n_{i-1}+1},\ d_{n_{i-1}+1})$, $(\sigma_{n_{i-1}+1},\ \rho_{n_{i-1}+1},\ \varrho_{n_{i-1}+1})$, $\ldots$, $(p_{n_i},\ c_{n_i},\ q_{n_i},\ r_{n_i},\ d_{n_i})$, each z_i denote

$(\epsilon,\ \rho_{n_i},\ \varrho_{n_i})$, each $\omega_i\dagger$ denote the trace $p_{n_{i-1}+1}$, $(\sigma_{n_{i-1}+1},\ \rho_{n_{i-1}+1})$, $p_{n_{i-1}+2}$, $(\sigma_{n_{i-1}+2},\ \rho_{n_{i-1}+2})$, ..., p_{n_i}, and each $\omega_i^{\ddagger}$ denote the trace $p_{n_{i-1}+1}$, $(\varrho_{n_{i-1}+1},\ \rho_{n_{i-1}+1})$, $p_{n_{i-1}+2}$, $(\varrho_{n_{i-1}+2},\ \rho_{n_{i-1}+2})$, ..., p_{n_i}. Also, for a trace $\omega = p_1, \sigma_1, \ldots, p_i, \sigma_i, p_{i+1}$, we denote its tail-removed subtrace $p_1, \sigma_1, \ldots, p_i, \sigma_i$ as ω^-.

5 Filter Generation

In addition to counterexample generation, one may further generate filters (also called vulnerability signatures in [31]) to block malicious input strings from the considered web application. By computing filters backward in the dependency graph, the filters for the input strings to an application can be obtained. The derived filters in our circuit representations are amenable for further hardware or firmware implementation to support a high-speed and low-power way of filtering malicious inputs from a web application. Notice that our circuit representation characterizes NFA in general, and further determinization may be needed for firmware or hardware implementation of filters. Although automata determinization can be costly, it is doable. Below we study how filter generation can be done under the proposed circuit representation.

First of all, the filter for the sink node of the dependency graph is available, assuming that sensitive strings to the underlying string manipulating program are known *a priori*. Moreover, consider an operator Op on a given set of input automata $A_1, \ldots, A_k$ yielding $A = \text{Op}(A_1, \ldots, A_k)$. Let B be an automaton with its language $\mathcal{L}(B) \subseteq \mathcal{L}(A)$ containing all illegal strings in $\mathcal{L}(A)$. We intend to construct the filter automaton B_i for some $i = 1, \ldots, k$ of concern such that $\mathcal{L}(B_i) \subseteq \mathcal{L}(A_i)$ and any $\boldsymbol{\sigma} \in \mathcal{L}(A_i)$ satisfies $(\mathcal{L}(\text{Op}(A_1, \ldots, A_{i-1}, A_{\boldsymbol{\sigma}}, A_{i+1}, \ldots, A_k)) \cap \mathcal{L}(B)) = \emptyset$ if and only if $\boldsymbol{\sigma} \notin \mathcal{L}(B_i)$, where $A_{\boldsymbol{\sigma}}$ denotes the automaton that accepts exactly the string $\boldsymbol{\sigma}$. Note that $\mathcal{L}(B_i)$ satisfying the above condition is a minimal filter provided that the relation among the inputs of an automata operation is ignored. Since the above condition guarantees that for each string in B_i, there exists a set of strings in other A_j's, $j \neq i$, such that some string in B is generated after apply Op on this set of strings of B_i and A_j's. Under the ignorance of the relation among the inputs of Op, a string should be kept in the language of filter automaton B_i as long as it may possibly result in a string in B through Op. The different Op cases are detailed in the following.

Intersection. Given the filter automaton B for the automaton $A = \text{Int}(A_1, A_2)$, the filter B can be directly applied as a filter for A_1 as well as A_2.

Union. Given the filter automaton B for $A = \text{Uni}(A_1, A_2)$, observe that every string in $\mathcal{L}(B)$ is in $\mathcal{L}(A_1)$ or in $\mathcal{L}(A_2)$. Hence automata $B_1 = \text{Int}(A_1, B)$ and $B_2 = \text{Int}(A_2, B)$ form legitimate filters for A_1 and A_2, respectively.

Concatenation. Given the filter automaton B for $A = \textsc{Cat}(A_1, A_2)$, to generate the corresponding filters B_1 and B_2 for A_1 and A_2, respectively, we first construct $B^\dagger = \textsc{Int}(A, B)$. Clearly, $\mathcal{L}(B^\dagger)$ equals $\mathcal{L}(B)$ because $\mathcal{L}(B) \subseteq \mathcal{L}(A)$. By the circuit construction of A, the auxiliary state variable α distinguishes between the substrings from $\mathcal{L}(A_1)$ and the substrings from $\mathcal{L}(A_2)$. As this information may not be seen in B, the purpose of this intersection is to identify the separation points between the two substring sources. Let B_1 be a copy of $B^\dagger$ but with the input symbol on every transition between states of $\alpha = 1$ being replaced with ϵ. Consider a trace (q_1, c_1), σ_1, ..., (q_i, c_i), ϵ, (q_{i+1}, c_{i+1}), ϵ, ..., (q_ℓ, c_ℓ) accepted by B_1, where $(q_j, c_j) \in [\![\boldsymbol{s}]\!]$ for $\boldsymbol{s}$ being the state variables of B_1, and $c_j \in [\![\alpha]\!]$ with $c_j = 0$ for $j \leq i$ and $c_j = 1$ for $j \geq i+1$. By the construction of B_1, there should be a trace (q_1, c_1), σ_1, ..., (q_i, c_i), ϵ, (q_{i+1}, c_{i+1}), σ_{i+1}, ..., (q_ℓ, c_ℓ) accepted by $B^\dagger$. The existence of such a trace ensures $\sigma_1\sigma_2 \ldots \sigma_{i-1} \in \mathcal{L}(A_1)$, $\sigma_{i+1}\sigma_{i+2} \ldots \sigma_{\ell-1} \in \mathcal{L}(A_2)$, and $\sigma_1\sigma_2 \ldots \sigma_{\ell-1} \in \mathcal{L}(B)$. The above trace accepted by $B^\dagger$ also ensures for each string, $\boldsymbol{\sigma} \in \mathcal{L}(B_1)$ if and only if there exists another string $\boldsymbol{\rho}$ in A_2 such that $\boldsymbol{\sigma}.\boldsymbol{\rho} \in \mathcal{L}(B)$. So B_1 forms a legitimate filter for A_1. Similarly, let B_2 be a copy of $B^\dagger$ but with the input symbol on every transition between states of $\alpha = 0$ being replaced with ϵ. Then B_2 forms a legitimate filter for A_2.

Replacement. Given the filter automaton B for $A = \textsc{Rep}(A_1, A_2, A_3)$, to generate the filter B_1 for automaton A_1, each string in $\mathcal{L}(B)$ has the form $\boldsymbol{\sigma}_1\boldsymbol{\tau}_1\boldsymbol{\sigma}_2\boldsymbol{\tau}_2 \ldots \boldsymbol{\sigma}_\ell$, where $\boldsymbol{\sigma}_i \in \overline{\Sigma^*.\mathcal{L}(A_2).\Sigma^*}$ and $\boldsymbol{\tau}_i \in \mathcal{L}(A_3)$ for $i = 1, \ldots, \ell$. We recognize each $\boldsymbol{\tau}_i$ and replace it with some string $\boldsymbol{\rho}_i \in \mathcal{L}(A_2)$. We then remove from the resultant language those strings not in A_1 by intersecting it with A_1. Therefore, B_1 can be constructed as follows.

First, similar to the construction of $A_1^{\lhd\rhd}$ in Sect. 3.5, we build automaton $B^{\lhd\rhd}$, which parenthesizes any substrings of a string in $\mathcal{L}(A_1)$. Second, similar to the construction of A_4 in Sect. 3.5, we build automaton B_4, which accepts the strings $\{(\boldsymbol{\sigma}_1. \lhd .\boldsymbol{\tau}_1. \rhd .\boldsymbol{\sigma}_2. \lhd .\boldsymbol{\tau}_2. \rhd \ldots) \in \Sigma^* \mid \boldsymbol{\sigma}_i \in \overline{\Sigma^*.\mathcal{L}(A_2).\Sigma^*}$ and $\boldsymbol{\tau}_i \in \mathcal{L}(A_3)\}$. Third, let $B_5 = \textsc{Int}(B^{\lhd\rhd}, B_4)$. Hence $\mathcal{L}(B_5) = \{(\boldsymbol{\sigma}_1. \lhd .\boldsymbol{\tau}_1. \rhd .\boldsymbol{\sigma}_2. \lhd .\boldsymbol{\tau}_2. \rhd \ldots) \in \Sigma^* \mid (\boldsymbol{\sigma}_1.\boldsymbol{\tau}_1.\boldsymbol{\sigma}_2.\boldsymbol{\tau}_2 \ldots) \in \mathcal{L}(B)$ and $\boldsymbol{\sigma}_i \in \overline{\Sigma^*.\mathcal{L}(A_2).\Sigma^*}$ and $\boldsymbol{\tau}_i \in \mathcal{L}(A_3)\}$. Then, in $\mathcal{L}(B_5)$ instead of replacing substrings $\lhd\mathcal{L}(A_3)\rhd$ with strings in $\mathcal{L}(A_2)$, we replace $\lhd$ with $\mathcal{L}(A_2)$, $\rhd$ with ϵ, and $\mathcal{L}(A_3)$ with ϵ. Let the resultant automaton be $B_1^\dagger$. Finally, $B_1 = \textsc{Int}(B_1^\dagger, A_1)$ forms a legitimate filter for A_1.

The fact that B_1 is a legitimate filter for A_1 can be shown as follows. Consider a string $\boldsymbol{\sigma} = \boldsymbol{\sigma}_1.\boldsymbol{\rho}_1.\boldsymbol{\sigma}_2.\boldsymbol{\rho}_2 \ldots \notin \mathcal{L}(B_1)$, where $\boldsymbol{\sigma}_i \notin \Sigma^*.\mathcal{L}(A_2).\Sigma^*$ and $\boldsymbol{\rho}_i \in \mathcal{L}(A_2)$. Also consider another string $\boldsymbol{\sigma}_1.\boldsymbol{\tau}_1.\boldsymbol{\sigma}_2.\boldsymbol{\tau}_2 \ldots$ obtained from replacing each $\boldsymbol{\rho}_i$ with $\boldsymbol{\tau}_i \in \mathcal{L}(A_3)$. If $\boldsymbol{\sigma}_1.\boldsymbol{\tau}_1.\boldsymbol{\sigma}_2.\boldsymbol{\tau}_2 \ldots \in \mathcal{L}(B)$, then we have $\boldsymbol{\sigma}_1. \lhd .\boldsymbol{\tau}_1. \rhd .\boldsymbol{\sigma}_2. \lhd .\boldsymbol{\tau}_2. \rhd \ldots \in \mathcal{L}(B^{\lhd\rhd})$. It is easy to see that $\boldsymbol{\sigma}_1. \lhd .\boldsymbol{\tau}_1. \rhd .\boldsymbol{\sigma}_2. \lhd .\boldsymbol{\tau}_2. \rhd \ldots \in \mathcal{L}(B_5)$. Finally, for each $\lhd.\boldsymbol{\tau}_i.\rhd$, replacing $\lhd$ with $\boldsymbol{\rho}_i$, replacing $\boldsymbol{\tau}_i$ with ϵ, and replacing $\rhd$ with ϵ yield $\boldsymbol{\sigma}_1.\boldsymbol{\rho}_1.\boldsymbol{\sigma}_2.\boldsymbol{\rho}_2 \ldots \in \mathcal{L}(B_1)$, which contradicts to the assumption $\boldsymbol{\sigma}_1.\boldsymbol{\rho}_1.\boldsymbol{\sigma}_2.\boldsymbol{\rho}_2 \ldots \notin \mathcal{L}(B_1)$. So we have $\mathcal{L}(\textsc{Rep}(A_{\boldsymbol{\sigma}}, A_2, A_3)) \cap \mathcal{L}(B) = \emptyset$ for any string $\boldsymbol{\sigma} \notin \mathcal{L}(B_1)$. Similarly, consider string $\boldsymbol{\sigma} = \boldsymbol{\sigma}_1.\boldsymbol{\rho}_1.\boldsymbol{\sigma}_2.\boldsymbol{\rho}_2 \ldots \in \mathcal{L}(B_1)$, where $\boldsymbol{\sigma}_i \notin \Sigma^*.\mathcal{L}(A_2).\Sigma^*$ and

$\boldsymbol{\rho}_i \in \mathcal{L}(A_2)$. Then it is in $\mathcal{L}(B_1^\dagger)$. By the construction of $B_1^\dagger$, there should be another string $\boldsymbol{\sigma}_1.\lhd.\boldsymbol{\tau}_1.\rhd.\boldsymbol{\sigma}_2.\lhd.\boldsymbol{\tau}_2.\rhd\ldots \in \mathcal{L}(B_5)$, where $\boldsymbol{\tau}_i \in \mathcal{L}(A_3)$. We have $\boldsymbol{\sigma}_1.\lhd.\boldsymbol{\tau}_1.\rhd.\boldsymbol{\sigma}_2.\lhd.\boldsymbol{\tau}_2.\rhd\ldots \in \mathcal{L}(B^{\lhd\rhd})$, and hence $\boldsymbol{\sigma}_1.\boldsymbol{\tau}_1.\boldsymbol{\sigma}_2.\boldsymbol{\tau}_2\ldots \in \mathcal{L}(B)$. It is easy to see that $\boldsymbol{\sigma}_1.\boldsymbol{\tau}_1.\boldsymbol{\sigma}_2.\boldsymbol{\tau}_2\ldots \in \mathcal{L}(\textsc{Rep}(A_{\boldsymbol{\sigma}}, A_2, A_3))$, which means $\mathcal{L}(\textsc{Rep}(A_{\boldsymbol{\sigma}}, A_2, A_3)) \cap \mathcal{L}(B) \neq \emptyset$. Consequently B_1 characterizes the desired language.

6 Extension to Symbolic Finite Automata

Symbolic finite automata (SFA) [26] extend conventional finite automata by allowing transition conditions to be specified in terms of predicates over a Boolean algebra with a potentially infinite domain. Formally, an SFA A is a 5-tuple $(Q, \mathcal{D}, I, \Delta, O)$, where Q is a finite set of states, $\mathcal{D}$ is the designated domain, $I \subseteq Q$ is the set of initial states (here we allow multiple initial states in contrast to the standard single-initial-state assumption of SFA), $\Delta : Q \times \Psi \times Q$ is the move relation for Ψ being the set of all quantifier-free formulas with at most one free variable, say χ, over a Boolean algebra of domain $\mathcal{D}$, $O \subseteq Q$ is the set of accepting states. We assume ϵ transitions are allowed and properly encoded in Δ in an SFA. Since $\mathcal{D}$ may not be bounded, a predicate logic formula over variable χ cannot be represented with logic circuits. We separate predicates from the logic circuit representation of SFA by abstracting each formula ψ appearing in Δ with its designated propositional variable x_ψ. Let $[\![\psi]\!]$ be extended to denote the set of solution values of χ satisfying ψ. Then the move relation of an SFA can be expressed with a transition relation

$$T(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}') = \bigvee_{(p,\psi,q)\in\Delta} (x_\psi \wedge (\boldsymbol{s} = p) \wedge (\boldsymbol{s}' = q))$$

and a predicate relation

$$P(\boldsymbol{x}, \chi) = \bigwedge_{(p,\psi,q)\in\Delta} (x_\psi \leftrightarrow (\chi \in [\![\psi]\!])).$$

Therefore we can represent an SFA A with four characteristic functions I, O, T, and P.

With the above construction, our circuit constructions of Sect. 3 naturally extend to SFA except that the predicate relation has to be additionally handled as follows. For SFA $A_{\text{INT}} = \textsc{Int}(A_1, A_2)$, the predicate relation

$$P_{\text{INT}}(\boldsymbol{x}, \chi) = P_1(\boldsymbol{x}_1, \chi) \wedge P_2(\boldsymbol{x}_2, \chi) \wedge (x_{\chi=\epsilon} \leftrightarrow \chi = \epsilon),$$

for $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, x_{\chi=\epsilon})$.

For SFA $A_{\text{UNI}} = \textsc{Uni}(A_1, A_2)$, the predicate relation

$$P_{\text{UNI}}(\boldsymbol{x}, \chi) = P_1(\boldsymbol{x}_1, \chi) \wedge P_2(\boldsymbol{x}_2, \chi),$$

for $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2)$.

For SFA $A_{\text{CAT}} = \text{CAT}(A_1, A_2)$, the predicate relation

$$P_{\text{CAT}}(\boldsymbol{x}, \chi) = P_1(\boldsymbol{x}_1, \chi) \wedge P_2(\boldsymbol{x}_2, \chi) \wedge (x_{\chi=\epsilon} \leftrightarrow \chi = \epsilon),$$

for $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, x_{\chi=\epsilon})$.

For SFA $A_{\text{REP}} = \text{REP}(A_1, A_2, A_3)$, we construct the predicate relation for A_{REP} as follows. The predicate relation of SFA $A_1^{\lhd\rhd}$ is first obtained from A_1 by

$$\begin{aligned} P_1^{\lhd\rhd}(\boldsymbol{x}_1^{\lhd\rhd}, \chi) = P_1(\boldsymbol{x}_1) \wedge (x_{\chi=\lhd} \leftrightarrow (\chi = \lhd)) \wedge (x_{\chi=\rhd} \leftrightarrow (\chi = \rhd)) \wedge \\ (x_{\chi\neq\lhd} \leftrightarrow (\chi \neq \lhd)) \wedge (x_{\chi\neq\rhd} \leftrightarrow (\chi \neq \rhd)), \end{aligned}$$

for $\boldsymbol{x}_1^{\lhd\rhd} = (\boldsymbol{x}_1, x_{\chi=\lhd}, x_{\chi=\rhd}, x_{\chi\neq\lhd}, x_{\chi\neq\rhd})$. Then the predicate relation of A_4 is constructed from those of A_2 and A_h by

$$\begin{aligned} P_4(\boldsymbol{x}_4, \chi) = P_2(\boldsymbol{x}_2, \chi) \wedge P_h(\boldsymbol{x}_h, \chi) \wedge (x_{\chi=\lhd} \leftrightarrow \chi = \lhd) \wedge (x_{\chi=\rhd} \leftrightarrow \chi = \rhd) \wedge \\ (x_{\chi\neq\lhd} \leftrightarrow \chi \neq \lhd) \wedge (x_{\chi\neq\rhd} \leftrightarrow \chi \neq \rhd), \end{aligned}$$

for $\boldsymbol{x}_4 = (\boldsymbol{x}_2, \boldsymbol{x}_h, x_{\chi=\lhd}, x_{\chi=\rhd}, x_{\chi\neq\lhd}, x_{\chi\neq\rhd})$. Then the predicate relation of A_5 is obtained by

$$P_5(\boldsymbol{x}_5, \chi) = P_1^{\lhd\rhd}(\boldsymbol{x}_1^{\lhd\rhd}, \chi) \wedge P_4(\boldsymbol{x}_4, \chi) \wedge (x_{\chi=\epsilon} \leftrightarrow \chi = \epsilon),$$

for $\boldsymbol{x}_5 = (\boldsymbol{x}_1^{\lhd\rhd}, \boldsymbol{x}_4, x_{\chi=\epsilon})$. Finally, the transition and predicate relations of SFA A_{REP} can be obtained by

$$\begin{aligned} T_{\text{REP}}(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}') = {} & (\neg\alpha \wedge \neg\alpha' \wedge T_5(\boldsymbol{x}_5, \boldsymbol{s}_5, \boldsymbol{s}_5') \wedge \neg\gamma \wedge \neg\gamma' \wedge I_3(\boldsymbol{s}_3) \wedge I_3(\boldsymbol{s}_3')) \vee \\ & (\neg\alpha \wedge \neg\alpha' \wedge (\boldsymbol{s}_5 = \boldsymbol{s}_5') \wedge (x_{\chi=\epsilon}) \wedge \neg\gamma \wedge \gamma' \wedge I_3(\boldsymbol{s}_3) \wedge I_3(\boldsymbol{s}_3')) \vee \\ & (\neg\alpha \wedge \neg\alpha' \wedge (\boldsymbol{s}_5 = \boldsymbol{s}_5') \wedge \gamma \wedge \gamma' \wedge T_3(\boldsymbol{x}_3, \boldsymbol{s}_3, \boldsymbol{s}_3')) \vee \\ & (\neg\alpha \wedge \alpha' \wedge T_5(\boldsymbol{x}_5, \boldsymbol{s}_5, \boldsymbol{s}_5')|_{\Delta[\chi/\lhd]} \wedge \gamma \wedge \neg\gamma' \wedge O_3(\boldsymbol{s}_3) \wedge I_3(\boldsymbol{s}_3') \wedge (x_{\chi=\epsilon})) \vee \\ & (\alpha \wedge \alpha' \wedge T_5(\boldsymbol{y}, \boldsymbol{s}_5, \boldsymbol{s}_5') \wedge (x_{\chi=\epsilon}) \wedge \neg\gamma \wedge \neg\gamma' \wedge I_3(\boldsymbol{s}_3) \wedge I_3(\boldsymbol{s}_3')) \vee \\ & (\alpha \wedge \neg\alpha' \wedge T_5(\boldsymbol{x}_5, \boldsymbol{s}_5, \boldsymbol{s}_5')|_{\Delta[\chi/\rhd]} \wedge \neg\gamma \wedge \neg\gamma' \wedge I_3(\boldsymbol{s}_3) \wedge I_3(\boldsymbol{s}_3') \wedge (x_{\chi=\epsilon})), \end{aligned}$$

$$P_{\text{REP}}(\boldsymbol{x}, \chi, \boldsymbol{y}, \chi^\dagger) = P_5(\boldsymbol{x}_5, \chi) \wedge P_3(\boldsymbol{x}_3, \chi) \wedge P_5(\boldsymbol{y}, \chi^\dagger),$$

where $\boldsymbol{x} = (\boldsymbol{x}_5, \boldsymbol{x}_3)$, $\boldsymbol{y}$ is a set of newly introduced propositional variables for $|\boldsymbol{x}|$, $\chi^\dagger$ is a newly introduced variable for χ serving for existential quantifications, and $T|_{\Delta[\chi/a]}$ denotes transition relation T is obtained under the modified move relation Δ in which variable χ is substituted with symbol a. (Here we avoid existentially quantifying out $\boldsymbol{y}$ and $\chi^\dagger$ by treating them as free variables.)

For emptiness checking of an SFA, we can treat the SFA as an infinite state transition system by considering $(\chi, \boldsymbol{x}, \boldsymbol{s})$ as the state variables. Let the transition relation be the conjunction of T and P, and let I and O be the initial and accepting state conditions, respectively, of the infinite state transition system. Then the model checking method [9], effectively PDR modulo theories, can be applied for reachability analysis.

7 Experimental Evaluation

Our tool, named SLOG, was implemented in the C language under the Berkeley logic synthesis and verification system ABC [6]. The experiments were conducted on a machine with Intel Xeon(R) 8-core CPU and 16 GB memory under the Ubuntu 12.04 LCS operating system.

We compared SLOG against other modern constraint solvers: CVC4 [3], Norn [1], Z3-str2 [34], and string analysis tools: JSA [8] and Stranger [30]. For the experiments, 20386 string analysis instances were generated from real web applications via Stranger [30]. The web applications includes Moodle, PHP-Fusion, etc., and these instances are tested for vulnerabilities such as SQL-injection, cross-site scripting (XSS), etc. Each instance corresponds to an acyclic dependency graph of a sink node in the program that consists of union, concatenation, and replacement operations. For each instance, we generated the string constraint that checks whether the dependency graph is vulnerable with respect to an attack pattern. String constraints were generated in the SMT-lib format for CVC4, Norn, and Z3-str2, and in the Java-program format for JSA.

The statistics of the benchmark instances are as follows. There are 85919 concatenation operations in total distributed in 18898 instances, 510 string replacement operations in 255 instances, and 25160 union operations in 5109 instances. All of these 20386 instances have membership checking at the end to determine whether an attack string can reach the sink node. All the solvers except for Norn, which does not support the replacement operation, provide full support on these string operations. Timeout limits 300 and 9000 s were set for small and large instances, respectively. An instance with fewer (resp. no fewer) than 100 concatenation operations is classified as small (resp. large).

The results of the solvers on the total 20386 instances are shown in Table 1, where #SAT, #UNS, #TO, #FL, and #Run denote the numbers of solved SAT, solved UNSAT, timeout, failed (with unexpected termination), and checked instances, respectively. The total runtimes for SAT and UNSAT instances are also shown in the table. Solvers SLOG, Stranger, CVC4, JSA and Z3-str2 checked all 20386 instances (runs) with successful rate 100 %, 100 %, 93.12 %, 99.98 %, 77.60 %, respectively; Norn checked 20131 instances with the successful rate 82.17 % without running the 255 instances with replacement operations.

Table 1. Statistics of solver performance

Solver	#SAT (time (s))	#UNS (time (s))	#TO	#FL	#Run
SLOG	8684 (65915)	11663 (72195)	39	0	20386
Stranger	8723 (10309)	11663 (1069)	0	0	20386
CVC4	7503 (8217)	11480 (1139)	1136	267	20386
JSA	8719 (7141)	11663 (8708)	0	4	20386
Z3-str2	4285 (249437)	11535 (921325)	2728	1838	20386
Norn	6306 (16586)	10236 (17383)	3344	245	20131

To evaluate solver performance on instances of different sizes, we classify the 20386 instances into three groups: the replacement-free small ones (with fewer than 100 concatenations and without replacement operations), the replacement-free large ones (with no fewer than 100 concatenations and without replacement operations), and the ones with replacement operations. By the classification, there are 20091 replacement-free small instances, 40 replacement-free large instances, and 255 instances with replacement operations. Note that the replacement-free large instances also have a large number of union operations.

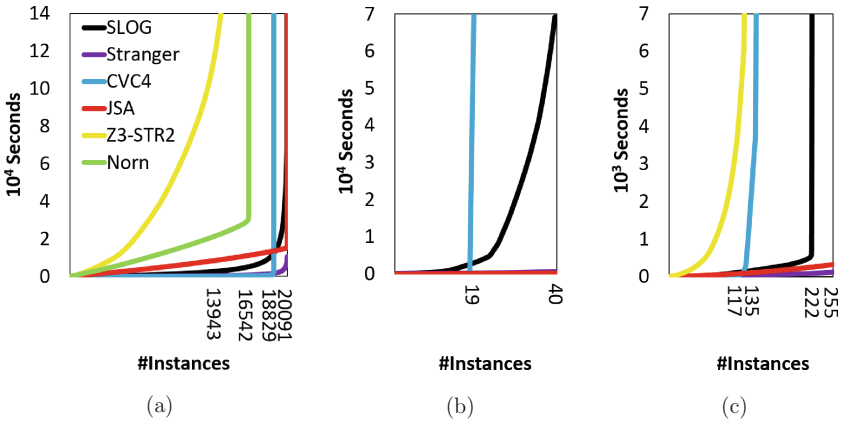

Fig. 2. Accumulated solving time for (a) replacement-free small instances, (b) replacement-free large instances, and (c) instances with replacement operations.

For the replacement-free small instances (under a 300-s timeout limit), the performances of solvers are shown in Fig. 2(a), where the x-axis is indexed by the number of solved instances, which are sorted by their runtimes in an ascending order for each solver, and the y-axis is indexed by the accumulated runtime in seconds. As shown in Fig. 2(a), SLOG successfully solves 20054 cases in 137670 seconds (with 37 timeout cases), outperforming Z3-STR2 (13943 cases in 1399712 s), CVC4 (18829 cases in 5555 s) and NORN (16542 cases in 33969 s). In contrast, STRANGER (20091 cases in 10590 s) and JSA (20087 cases in 15336 s) outperform SLOG on almost all the cases. For the replacement-free large instances (under a 9000-s timeout limit), both Z3-STR2 and NORN failed to solve any due to timeout. As seen from Fig. 2(b), SLOG solved most of the large cases (with an average of 1750 s per case), while CVC4 solved fewer than half of the instances (19 out of 40) but took less time on solvable instances. STRANGER and JSA outperform SLOG and other SMT-based solvers, being able to solve all the 40 cases with less time. For the instances with replacement operations (under a 300-s timeout limit), all solvers are applicable except for NORN. Figure 2(c) shows that the relative performances of the solvers are similar to those in the other two instance groups. The reason that STRANGER outperforms SLOG might be explained by

the fact that the emptiness checking of a sink automaton in Stranger is of constant time complexity (due to the canonicity of state-minimized DFA), while that in SLOG requires reachability analysis. Therefore as long as Stranger succeeds in building the sink automaton, it is likely to outperform SLOG.

With the auxiliary variables and other information embedded in the circuit construction, SLOG can generate counterexamples. We applied SLOG to find witnesses of all 8684 vulnerable instances. It took 524 s in total to generate counterexamples for all 8684 instances, only a small fraction of the total constraint solving time 65915 s. The high efficiency of counterexample generation in SLOG can be attributed to the fact that the assignments to the internally quantified variables in our circuit construction are already computed by PDR. There is no need to re-derive them in generating counterexample traces by the rules of Sect. 4 (Table 2).

Table 2. SLOG performance on counterexample generation

Group	#SAT	SolveTime (s)	CexGenTime (s)
Small	8426	60664	481
Large	22	4236	18
Replacement	236	1015	25

In summary, SLOG performed the best among the solvers with counterexample generation capability, including CVC4, Z3-str2 and Norn. In fact, a significant portion of runtime spent by SLOG is on running PDR for language emptiness checking. Although Stranger and JSA performed better than SLOG in runtime, both are incapable of finding as a witness the values of input nodes to a specific attack string in the sink node.

To justify that our circuit-based method can be more scalable than BDD-based methods for representing automata with large alphabets, consider the automata over alphabet $\Sigma \times \Sigma$ with $|\Sigma| = 2^n$ accepting the language $(a, a)^*$ for $a \in \Sigma$. The automata have a linear $O(n)$ AIG representation ($4n+1$ gates), but have an exponential $O(2^n)$ BDD representation (e.g., 46 BDD nodes for $n = 4$, 766 nodes for $n = 8$, and 196606 nodes for $n = 16$) in MONA [7], which is used by Stranger. Although a good BDD variable ordering exists to reduce the BDD growth rate to linear in this example, a good BDD variable ordering can be hard to find and even may not exist in general. In addition, because SLOG represents NFA instead of DFA, it may avoid costly subset construction and can be more compact than (DFA-based) Stranger.

8 Discussions

While SLOG demonstrates its ability on string constraint solving and counterexample generation by taking advantage of circuit-based NFA representation,

it should be noted that the compared string analysis tools have varied focuses and expressiveness of specifying (non)string constraints. CVC4 [3] is a SMT-based solver that supports many-sorted first-order logic. Norn [1] is another SMT-based string constraint solver that employs Craig interpolation to handle word equations over (unbounded length) string variables, constraints of string length, and regular language membership constraints. Z3-str2 [34] is a string theory plug-in built upon SMT solver Z3 [22]. These string solvers address string constraints with lengths and can generate witness for satisfying constraints. In the experimental evaluation, we did not consider length constraints when generating dependency graphs. String constraints with lengths are not currently supported by SLOG. The circuit-based representation could be extended to model arithmetic automata for automata-based string-length constraint solving [2,33]. JSA is an explicit automata tool for analyzing the flow of strings and string operations in Java programs. Stranger is an MTBDD-based automata library for symbolic string analysis, which can be used to solve string constraints and compute pre- and post-images of string manipulation operations. JSA employes grammatical string analysis with regular language approximation and incorporates finite state transducers to support language-based replacement operations, while Stranger can conduct forward and backward reachability analysis of string manipulation programs along with DFA constructions for language operations. In the evaluation, we did not conduct analysis on cyclic dependency graphs that can be analyzed with JSA and Stranger. Conducting fixpoint computation on cyclic dependency graphs may require efficient complement operation in our circuit-based NFA representation that is not currently supported by SLOG.

9 Conclusions

We have presented a circuit-based NFA manipulation package for string analysis. Compared to BDD-based methods of automata representation, our circuit-based representation is scalable to automata with large alphabets. Our method avoids costly determinization whenever possible. It supports both counterexample generation and filter synthesis. In addition, extension to symbolic finite automata has been shown. Experiments have shown the unique benefits of our method. For future work, it would be interesting to explore the usage of SLOG as a string analysis engine in SMT solvers.

Acknowledgments. This work was supported in part by the Ministry of Science and Technology of Taiwan under grants MOST 104-2628-E-002-013-MY3, 104-2218-E-001-002, and 103-2221-E-004-006-MY3.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: an SMT solver for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 462–469. Springer, Heidelberg (2015)

2. Aydin, A., Bang, L., Bultan, T.: Automata-based model counting for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 255–272. Springer, Heidelberg (2015)
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011)
4. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 307–321. Springer, Heidelberg (2009)
5. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
6. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010)
7. BRICS: The MONA project. http://www.brics.dk/mona/
8. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
9. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 46–61. Springer, Heidelberg (2014)
10. D'Antoni, L., Veanes, M.: Extended symbolic finite automata and transducers. Formal Meth. Syst. Des. **47**(1), 93–119 (2015)
11. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: FMCAD, pp. 125–134 (2011)
12. Gould, C., Su, Z., Devanbu, P.: Static checking of dynamically generated queries in database applications. In: ICSE, pp. 645–654 (2004)
13. Hooimeijer, P., Weimer, W.: StrSolve: solving string constraints lazily. Autom. Softw. Eng. **19**(4), 531–559 (2012)
14. Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.: Securing web application code by static analysis and runtime protection. In: WWW, pp. 40–52 (2004)
15. Jensen, S.H., Jonsson, P.A., Møller, A.: Remedying the eval that men do. In: ISSTA, pp. 34–44 (2012)
16. Jiang, J.H.R., Brayton, R.K.: On the verification of sequential equivalence. IEEE Trans. Comp. Aid. Des. Int. Circ. Syst. **22**(6), 686–697 (2003)
17. Jovanovic, N., Krügel, C., Kirda, E.: Pixy: a static analysis tool for detecting web application vulnerabilities. In: S&P, pp. 258–263 (2006)
18. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: a solver for string constraints. In: ISSTA, pp. 105–116 (2009)
19. Li, G., Ghosh, I.: PASS: string solving with parameterized array and interval automaton. In: Bertacco, V., Legay, A. (eds.) HVC 2013. LNCS, vol. 8244, pp. 15–31. Springer, Heidelberg (2013)
20. Minamide, Y.: Static approximation of dynamically generated web pages. In: WWW, pp. 432–441 (2005)
21. Mishchenko, A., Chatterjee, S., Jiang, J.H.R., Brayton, R.: FRAIGs: a unifying representation for logic synthesis and verification. In: ERL Technical report, UC Berkeley (2005)

22. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
23. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: S&P, pp. 513–528 (2010)
24. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: POPL, pp. 372–382 (2006)
25. Veanes, M.: Applications of symbolic finite automata. In: Konstantinidis, S. (ed.) CIAA 2013. LNCS, vol. 7982, pp. 16–23. Springer, Heidelberg (2013)
26. Veanes, M., de Halleux, P., Tillmann, N.: Rex: symbolic regular expression explorer. In: ICST, pp. 498–507 (2010)
27. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjørner, N.: Symbolic finite state transducers: algorithms and applications. In: POPL, pp. 137–150 (2012)
28. Veanes, M., Mytkowicz, T., Molnar, D., Livshits, B.: Data-parallel string-manipulating programs. In: POPL, pp. 139–152 (2015)
29. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: PLDI, pp. 32–41 (2007)
30. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: an automata-based string analysis tool for PHP. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 154–157. Springer, Heidelberg (2010)
31. Yu, F., Alkhalaf, M., Bultan, T.: Patching vulnerabilities with sanitization synthesis. In: ICSE, pp. 251–260 (2011)
32. Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. Formal Meth. Syst. Des. **44**(1), 44–70 (2014)
33. Yu, F., Bultan, T., Ibarra, O.H.: Symbolic string verification: combining string analysis and size analysis. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 322–336. Springer, Heidelberg (2009)
34. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Dolby, J., Zhang, X.: Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 235–254. Springer, Heidelberg (2015)