Model-Checking Software Library API Usage Rules*

Fu Song and Tayssir Touili

LIAFA, CNRS and Univ. Paris Diderot, France
{song,touili}@liafa.univ-paris-diderot.fr

Abstract. Modern software increasingly relies on using libraries which are accessed via Application Programming Interfaces (APIs). Libraries usually impose constraints on how API functions can be used (API usage rules) and programmers have to obey these API usage rules. However, API usage rules often are not well-documented or documented informally. In this work, we show how to use the SCTPL logic to precisely specify API usage rules in libraries, where SCTPL can be seen as an extension of the branching-time temporal logic CTL with variables, quantifiers, and predicates over the stack. This allows library providers to formally describe API usage rules without knowing how their libraries will be used by programmers. We also propose an approach to automatically check whether programs using libraries violate or not the corresponding API usage rules. Our approach consists in modeling programs as pushdown systems (PDSs), and checking API usage rules on programs using SCTPL model checking for PDSs. To make the model-checking procedure more efficient, we propose an abstraction that reduces drastically the size of the program model. Moreover, we characterize a sub-logic rSCTPL of SCTPL preserved by the abstraction. rSCTPL is sufficient to precisely specify all the API usage rules we met. We implemented our techniques in a tool and applied it to check several API usage rules. Our tool detected several previously unknown errors in well-known programs, such as Nssl, Verbs, Acacia+, Walksat and Getafix. Our experimental results are encouraging.

1 Introduction

Most modern software increasingly relies on using libraries and frameworks provided by organizations in order to shorten time to market. Libraries or frameworks are accessed via Application Programming Interfaces (APIs) which are sets of library functions (called API functions) and usually impose constraints (API usage rules) on how API functions can be used. Programmers have to obey these constraints when calling API functions. However, most of API usage rules are not well-documented or documented informally in the API documentation. It is easy to introduce bugs using API functions. So, it is important to formally describe and automatically check API usage rules.

Many works addressed this problem [15, 19, 22, 24–26, 28, 30, 32–36, 38, 44, 45, 47]. However, their approaches either cannot describe API usage rules in a precise manner or cannot automatically check API usage rules. In this paper, we propose a novel technique to specify and check API usage rules without knowing how API functions will be

* Work partially funded by ANR grant ANR-08-SEGI-006.

E.B. Johnsen and L. Petre (Eds.): IFM 2013, LNCS 7940, pp. 192–207, 2013.

used by programmers. Our approach consists of (1) modeling programs as pushdown systems (PDSs), since PDSs are a natural model of sequential programs [23] (the stack of PDSs stores the calling procedures which allows us to check context-sensitive API usage rules), (2) specifying in a precise manner API usage rules in the Stack Computation Tree Predicate Logic (SCTPL) [41] (indeed, SCTPL can describe several API usage rules that cannot be expressed by the existing works), and (3) automatically checking whether programs violate or not API usage rules by SCTPL model checking for PDSs.

SCTPL can be seen as an extension of the CTPL logic with predicates over the stack content. CTPL [29] is an extension of the Computation Tree Logic (CTL) with variables and quantifiers. In CTPL, propositions can be predicates of the form $p(x_1, ..., x_m)$, where the x_i's are free variables or constants. Free variables can get their values from a finite domain and be universally or existentially quantified. CTPL can specify API usage rules without knowing how API functions will be used by programmers. E.g., consider the file operation API usage rule "The file should be closed by calling the API function *fclose* whenever this file is opened by calling *fopen*". Closing opened files is important. Indeed, long time running programs, such as web servers, will use many resources if opened files are not closed. This API usage rule can be expressed in CTL as $\psi_1 \equiv \mathbf{AG}(fopen \Longrightarrow \mathbf{EF}fclose)$ (note that the formula $\psi_1' \equiv \mathbf{AG}(fopen \Longrightarrow \mathbf{AF}fclose)$ is incorrect, if *fopen* returns a *null* file pointer, then *fclose* should not be called). However, ψ_1 cannot detect the bug in Figure 2, where the file pointed to by f_2 is never closed. This is due to the fact that we cannot specify the relation between the return value of *fopen* and the parameter of *fclose*. To detect this bug, we have to specify this rule as $\psi_2 \equiv \mathbf{AG}(\bigwedge_{i=1}^{3}(f_i = fopen \Longrightarrow \mathbf{EF}fclose(f_i)))$. However, this formula is too special to specify this rule in library, since e.g., replacing the variable f_1 by f_1' breaks ψ_2. Using CTPL, we can specify this rule as $\psi_3 \equiv \forall x \forall y \forall z\ \mathbf{AG}(x = fopen(y, z) \Longrightarrow \mathbf{EF}fclose(x))$ stating that whenever a file is opened and pointed to by some variable x, it should be closed in the future. [1]

```
n1 : FILE* f1=fopen("t1","w");
n2 : FILE* f2=fopen("t2","w");
n3 : FILE* f3=fopen("t3","w");
n4 : if(f1) then
n5 :   fclose(f1);
n6 : fclose(f3);
```

Fig. 2. File Operations

However, CTPL cannot specify properties about the calling procedures. Being able to express such properties is important. E.g., consider an API usage rule expressing that "Calling a function $proc_1$ in some procedure *proc* must be followed by a call to the function $proc_2$ before the procedure *proc* returns". This API usage rule cannot be specified in CTPL. To overcome this problem, we use the SCTPL logic [40, 41] to precisely describe API usage rules. SCTPL extends CTPL by predicates over the stack. Such predicates are given by regular expressions over the stack alphabet and some free

[1] Note that ψ_3 cannot express the point that *fclose* is only called when *fopen* returns a pointer to the file. Indeed, *fopen* returns a null pointer when the file does not exist. In this case, calling *fclose(*f_3*)* induces an error. To express such a point, we introduce an additional predicate *Test(x)* which holds at some control point n iff x is tested at the control point n. Now, we can refine the rule into $\psi_4 \equiv \forall x \forall y \forall z\ \mathbf{AG}(x = fopen(y, z) \Longrightarrow \mathbf{AF}(Test(x) \wedge \mathbf{EXAF}fclose(x)))$. ψ_4 states that whenever $x = fopen$ is made, one has to check the return value x (i.e., *Test(x)*). After this, the file has to be closed in all the future paths. The motivation of using *Test(x)* is that we cannot know how the return value will be checked. Thus, we coarsely specify that the return value is checked.

variables (which can also be existentially and universally quantified). Using SCTPL, the above rule can be specified as $\forall l\ \mathbf{AG}((proc_1 \wedge \Gamma l \Gamma^*) \Longrightarrow \mathbf{AF}(proc_2 \wedge \Gamma^+ l \Gamma^*))$, where $\Gamma l \Gamma^*$ and $\Gamma^+ l \Gamma^*$ are regular predicates. The subformula $(proc_1 \wedge \Gamma l \Gamma^*)$ expresses that $proc_1$ is called inside some procedure *proc* whose return address is l (since the return addresses of the called procedures are put into the stack when executing the program.). The above formula states that whenever $proc_1$ is called in some procedure *proc* whose return address is l (ensured by $\Gamma l \Gamma^*$), a function call to $proc_2$ should be made where the return address l is still in the stack, i.e., before the procedure *proc* returns (this is ensured by $\Gamma^+ l \Gamma^*$). Note that, in our modeling, the topmost symbol of the stack of the PDS stores the current control point, the rest of the stack stores the return addresses of the calling procedures, i.e., the procedures that have not returned yet.

It is shown in [41] that SCTPL model checking for PDSs is decidable. Thus, we can automatically check whether a program violates or not API usage rules by SCTPL model-checking for PDSs. To make the verification of API usage rules more efficient, we introduce the *procedure-cutting abstraction*, which is an abstraction that drastically reduces the size of the program model by removing some procedures that do not use the API functions specified in the SCTPL formula. We also consider rSCTPL, a sub-logic of SCTPL and show that the procedure-cutting abstraction preserves all rSCTPL formulas when the removed procedures are infinite execution free. rSCTPL is sufficient to express all the API usage rules we met. Moreover, rSCTPL can describe all API usage rules we met. Our abstraction allowed us to apply our techniques to large programs.

The main contributions of this paper are:

1. We propose a novel approach to precisely specify API usage rules using SCTPL. SCTPL allows library providers to formally describe API usage rules when implementing the libraries.
2. We can automatically check programs against API usage rules by SCTPL model-checking. Our techniques also allow program developers to automatically verify API usage rules of their programs without any additional inputs nor environment abstractions.
3. We propose a procedure-cutting abstraction. We show that this abstraction preserves all rSCTPL formulas when the cut procedures are infinite execution free. Our abstraction reduces drastically the size of the program model, which makes API usage rules verification more efficient.
4. We implemented our techniques in a tool and applied it to check several API usage rules on several open source programs. Our tool was able to find several unknown bugs in some well-known open source programs, such as Nssl, Verbs, Acacia+, Walksat and Getafix.

Outline. Section 2 gives a formal definition of PDSs. Section 3 recalls the definition of SCTPL, and shows how to precisely specify API usage rules in SCTPL. Section 4 describes the procedure-cutting abstraction and the sub-logic rSCTPL of SCTPL. Section 5 discusses the experimental results. The related work is given in Section 6.

2 Formal Model: Pushdown Systems

In this section, we recall the definition of pushdown systems. We use the approach of [23] to model a sequential program as a pushdown system.

A *Pushdown System* (PDS) is a tuple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of control locations, Γ is the stack alphabet, $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of transition rules. A configuration $\langle p, \omega \rangle$ of $\mathcal{P}$ is an element of $P \times \Gamma^*$. We write $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$ instead of $((p, \gamma), (q, \omega)) \in \Delta$. The successor relation $\rightsquigarrow_{\mathcal{P}} \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ is defined as follows: if $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$, then $\langle p, \gamma\omega' \rangle \rightsquigarrow_{\mathcal{P}} \langle q, \omega\omega' \rangle$ for every $\omega' \in \Gamma^*$. A *path* of the PDS is a sequence of configurations $c_1 c_2 ...$ such that c_{i+1} is an *immediate successor* of the configuration c_i, i.e., $c_i \rightsquigarrow_{\mathcal{P}} c_{i+1}$, for every $i \geq 1$.

3 API Usage Rules Specification

In this section, we recall the definition of the Stack Computation Tree Predicate Logic (SCTPL) [41], and show how to specify API usage rules in SCTPL.

3.1 Environments, Predicates and Regular Variable Expressions

Hereafter, we fix the following notations. Let $\mathcal{X} = \{x_1, x_2, ...\}$ be a finite set of variables ranging over a finite domain $\mathcal{D}$. Let $B : \mathcal{X} \cup \mathcal{D} \longrightarrow \mathcal{D}$ be an environment function that assigns a value $v \in \mathcal{D}$ to each variable $x \in \mathcal{X}$ and such that $B(v) = v$ for every $v \in \mathcal{D}$. $B[x \leftarrow v]$ denotes the environment function such that $B[x \leftarrow v](x) = v$ and $B[x \leftarrow v](y) = B(y)$ for every $y \neq x$. Let $\mathcal{B}$ be the set of all the environment functions.

Let AP be a finite set of atomic propositions, $AP_{\mathcal{X}}$ be a finite set of atomic predicates in the form of $a(\alpha_1, ..., \alpha_m)$ such that $a \in AP$, $\alpha_i \in \mathcal{X} \cup \mathcal{D}$ for every $1 \leq i \leq m$, and $AP_{\mathcal{D}}$ be a finite set of atomic predicates of the form $a(\alpha_1, ..., \alpha_m)$ such that $a \in AP$, $\alpha_i \in \mathcal{D}$ for every $1 \leq i \leq m$.

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, let $\mathcal{R}$ be a finite set of regular variable expressions over $\mathcal{X} \cup \Gamma$ given by: $e ::= \emptyset \mid \epsilon \mid a \in \mathcal{X} \cup \Gamma \mid e + e \mid e \cdot e \mid e^*$.

The language $L(e)$ of a regular variable expression e is a subset of $P \times \Gamma^* \times \mathcal{B}$ defined inductively as follows: $L(\emptyset) = \emptyset$; $L(\epsilon) = \{(\langle p, \epsilon \rangle, B) \mid p \in P, B \in \mathcal{B}\}$; $L(x)$, where $x \in \mathcal{X}$ is the set $\{(\langle p, \gamma \rangle, B) \mid p \in P, \gamma \in \Gamma, B \in \mathcal{B} : B(x) = \gamma\}$; $L(\gamma)$, where $\gamma \in \Gamma$ is the set $\{(\langle p, \gamma \rangle, B) \mid p \in P, B \in \mathcal{B}\}$; $L(e_1 + e_2) = L(e_1) \cup L(e_2)$; $L(e_1 \cdot e_2) = \{(\langle p, \omega_1\omega_2 \rangle, B) \mid (\langle p, \omega_1 \rangle, B) \in L(e_1);\ (\langle p, \omega_2 \rangle, B) \in L(e_2)\}$; and $L(e^*) = \{(\langle p, \omega \rangle, B) \mid B \in \mathcal{B}$ and $\omega = \omega_1 \cdots \omega_m$, s.t. $\forall i, 1 \leq i \leq m, (\langle p, \omega_i \rangle, B) \in L(e)\}$. E.g., $(\langle p, \gamma_1\gamma_2\gamma_2 \rangle, B)$ is an element of $L(\gamma_1 x^*)$ when $B(x) = \gamma_2$.

3.2 Stack Computation Tree Predicate Logic

A SCTPL formula is a CTL formula where predicates and regular variable expressions are used as atomic propositions and variables can be quantified. Regular variable expressions are used to express predicates on the stack content of the PDS. More precisely, the set of *SCTPL formulas* is given by (where $x \in \mathcal{X}$, $a(x_1, ..., x_m) \in AP_{\mathcal{X}}$ and $e \in \mathcal{R}$):

$$\varphi ::= a(x_1, ..., x_m) \mid e \mid \neg\varphi \mid \varphi \wedge \varphi \mid \forall x\, \varphi \mid \mathbf{EX}\varphi \mid \mathbf{EG}\varphi \mid \mathbf{E}[\varphi \mathbf{U} \varphi].$$

Let φ be a SCTPL formula. The closure $cl(\varphi)$ denotes the set of all the subformulas of φ including φ.

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ s.t. $\Gamma \subseteq \mathcal{D}$, let $\lambda : AP_{\mathcal{D}} \rightarrow 2^{\Gamma}$ be a labeling function that assigns a set of stack symbols to a predicate. Let $c \in P \times \Gamma^*$ be a configuration of $\mathcal{P}$. $\mathcal{P}$ satisfies a SCTPL formula ψ in c, denoted by $c \models_{\lambda} \psi$, iff there exists an environment $B \in \mathcal{B}$ s.t. $c \models^B_{\lambda} \psi$, where $c \models^B_{\lambda} \psi$ is defined by induction as follows:

- $c \models^B_{\lambda} a(x_1, ..., x_m)$ iff $n \in \lambda(a(B(x_1), ..., B(x_m)))$ and $c = \langle p, n\omega \rangle$.
- $c \models^B_{\lambda} e$ iff $(c, B) \in L(e)$.
- $c \models^B_{\lambda} \psi_1 \wedge \psi_2$ iff $c \models^B_{\lambda} \psi_1$ and $c \models^B_{\lambda} \psi_2$.
- $c \models^B_{\lambda} \forall x\, \psi$ iff $\forall v \in \mathcal{D},\ c \models^{B[x \leftarrow v]}_{\lambda} \psi$.
- $c \models^B_{\lambda} \neg\psi$ iff $c \not\models^B_{\lambda} \psi$.
- $c \models^B_{\lambda} \mathbf{EX}\, \psi$ iff there exists a successor c' of c s.t. $c' \models^B_{\lambda} \psi$.
- $c \models^B_{\lambda} \mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ iff there exists a path $\pi = c_0 c_1 ...$ of $\mathcal{P}$ with $c_0 = c$ s.t. $\exists i \geq 0,\ c_i \models^B_{\lambda} \psi_2$ and $\forall 0 \leq j < i, c_j \models^B_{\lambda} \psi_1$.
- $c \models^B_{\lambda} \mathbf{EG}\psi$ iff there exists a path $\pi = c_0 c_1 ...$ of $\mathcal{P}$ with $c_0 = c$ s.t. $\forall i \geq 0$: $c_i \models^B_{\lambda} \psi$.

Intuitively, $c \models^B_{\lambda} \psi$ holds iff the configuration c satisfies ψ under the environment B. We will freely use the following abbreviations: $\mathbf{AX}\psi = \neg\mathbf{EX}(\neg\psi)$, $\mathbf{EF}\psi = \mathbf{E}[true\mathbf{U}\psi]$, $\mathbf{AG}\psi = \neg\mathbf{EF}(\neg\psi)$, $\mathbf{AF}\psi = \neg\mathbf{EG}(\neg\psi)$, $\mathbf{A}[\psi_1\mathbf{U}\psi_2] = \neg\mathbf{E}[\neg\psi_2\mathbf{U}(\neg\psi_1 \wedge \neg\psi_2)] \wedge \neg\mathbf{EG}\neg\psi_2$, $\mathbf{A}[\psi_1\mathbf{R}\psi_2] = \neg\mathbf{E}[\neg\psi_1\mathbf{U}\neg\psi_2]$, $\mathbf{E}[\psi_1\mathbf{R}\psi_2] = \neg\mathbf{A}[\neg\psi_1\mathbf{U}\neg\psi_2]$, and $\exists x\psi = \neg\forall x\neg\psi$.

Theorem 1. *[41] SCTPL model-checking for PDSs is decidable.*

3.3 Extracting Predicates for API Specifications

API usage rules often state properties concerning the order of API function calls and return value tests. Indeed, usually, after making a call to an API function, one has to check whether the call was successful. For example, when *fopen* is called to open a file *t1*, one has to make sure that the call was successful, i.e., that the file *t1* exists (as done in Figure 2, Line n_4). Thus, to check API usage rules, we need to extract predicates about API function calls and return value tests. To do this, for every API function call $y = f(p_1, ..., p_m)$ at a control point n where y denotes the return value[2] and for every $1 \leq i \leq m$, p_i denotes the i^{th} parameter of the function f, we add the predicate $f(p_1, ..., p_m, y)$ to $AP_{\mathcal{D}}$ and associate this predicate to the control point n (i.e., we let $n \in \lambda(f(p_1, ..., p_m, y))$). By abuse of notation, such predicates $f(p_1, ..., p_m, y)$ will also be denoted by $y = f(p_1, ..., p_m)$.

For every boolean expression b in a conditional statement (e.g., if-then-else, switch-case) at a control point n s.t. y is used in b and y is a return value of some function call, we add the predicate *Test(y)* in $AP_{\mathcal{D}}$ and associate this predicate to n (i.e., we let $n \in \lambda(Test(y))$).

Intuitively, for every $\omega \in \Gamma^*$, a configuration $\langle s_0, n\omega \rangle$ satisfies the atomic predicate σ (i.e., σ is $y = f(p_1, ..., p_m)$ or *Test(x)*) iff σ is associated to n (i.e., $n \in \lambda(\sigma)$). W.l.o.g., we suppose that the return value of some API function is immediately checked in the same procedure where the API function is called. This assumption will not restrict the usefulness of the libraries, and it is recommended to check the return value immediately after the function call.

[2] W.l.o.g., we assume that each function call has a return value assigned to some variable.

$\lambda(f_1 = fopen(\text{“}t_1\text{”}, \text{“}w\text{”})) = \{n_1\}$	$\langle s_0, n_1\rangle \hookrightarrow \langle s_0, fo_0n_2\rangle$
$\lambda(f_2 = fopen(\text{“}t_2\text{”}, \text{“}w\text{”})) = \{n_2\}$	$\langle s_0, n_2\rangle \hookrightarrow \langle s_0, fo_0n_3\rangle$
$\lambda(f_3 = fopen(\text{“}t_3\text{”}, \text{“}w\text{”})) = \{n_3\}$	$\langle s_0, n_3\rangle \hookrightarrow \langle s_0, fo_0n_4\rangle$
$\lambda(Test(f_1)) = \{n_4\}$	$\langle s_0, n_4\rangle \hookrightarrow \langle s_0, n_5\rangle$
$\lambda(fclose(f_1)) = \{n_5\}$	$\langle s_0, n_5\rangle \hookrightarrow \langle s_0, fc_0n_6\rangle$
$\lambda(fclose(f_3)) = \{n_6\}$	$\langle s_0, n_6\rangle \hookrightarrow \langle s_0, fc_0n_7\rangle$
(a)	(b)

Fig. 2. (a) The labeling function λ and (b) Transition rules Δ

3.4 An Illustrating Example

To illustrate our approach, we show how to specify the API usage rules for the GNU socket library.

Description of the Socket Library The socket library implements a generalized interprocess communication channel. It provides TCP and UDP Protocols. As shown in Figure 3, a server-side program using the TCP Protocol should first create a socket *s* by calling *socket* with SOCK_STREAM as second parameter, then bind *s* to some address by calling *bind* and listen to the address by calling *listen*. When the server receives a connection request, it will create a new socket *ns* by calling *accept*. Then, the server can communicate with the client by calling *send* and *recv* via the socket *ns*. Finally, *s* and *ns* should be destroyed by calling *close*.

Figure 4 shows a typical application of the TCP Protocol at the client-side. It connects to a server by calling *connect* after creating the socket *s*. Then, it can communicate with the server by calling *send* and *recv* via the socket *s*. Finally, *s* should be destroyed by calling *close*.

The server-side program using the UDP Protocol should create a socket *s* by calling *socket* with SOCK_DGRAM as second parameter as shown in Figure 5. After that, it should bind *s* to some address by calling *bind*. Then, it can communicate with a client by calling *recvfrom* and *sendto* via *s*. Finally, the socket *s* should be closed by calling *close*. The client-side program using the UDP Protocol can communicate with a server by calling *recvfrom* and *sendto* via a socket *s* after its creation. Figure 6 is a typical implementation of the UDP Protocol at the client-side.

```
1  int s, c, ns;
2  if((s=socket(AF_INET,SOCK_STREAM,0))==-1)
3    return;
4  if(bind(s,&s_addr,len)==-1)
5    {close(s); return;}
6  if(listen(s,5)==-1){close(s); return;}
7  while(1){
8    ns=accept(s,&c_addr, &size);
9    do{
10     recv(ns,data,256,0);
11     ...
12     send(ns,data,256,0);
13     if(cond1){close(ns); return;}
14   }while(cond2)
15 }
16 close(s);
```

Fig. 3. TCP Server-side

```
1  int s;
2  if((s=socket(AF_INET,SOCK_STREAM,
3                0))==-1)
4    return;
5  ...
6  connect(s,&s_addr,len)
7  do{
8    send(s,data,256,0);
9    ...
10   recv(s,data,256,0);
11 }while(cond3)
12 close(s);
```

Fig. 4. TCP Client-side

```
1  int s;
2  if((s=socket(AF_INET, SOCK_DGRAM,0))==-1)
3    return;
4  if(bind(s,&s_addr,sizeof(s_addr))==-1)
5    { close(s); return; }
6  do{
7    recvfrom(s,data,256,0,&c_addr, len);
8    sendto(s,data,256,0,&c_addr,len);
9  }while(cond4)
10 close(s);
```

Fig. 5. UDP Server-side

```
1  int s;
2  if((s=socket(AF_INET,SOCK_DGRAM,
3                0))==-1)
4    return;
5  do(1){
6    sendto(s,data,256,0,&addr, len);
7    ...
8    recvfrom(s,data,256,0,&addr, len);
9  }while(cond5)
10 close(s);
```

Fig. 6. UDP Client-side

Specifying the Socket Library API Usage Rules in SCTPL. Table 1 shows some SCTPL formulas describing some API usage rules of the socket library. Let us consider the API usage rule "The return value of *socket* should be checked immediately after the call to *socket* is made, and after a socket is created, this socket should be destroyed in all the future paths". We can specify this rule by the SCTPL formula r_1 as shown in Table 1. r_1 states that whenever the call to *socket* is made in a procedure *proc* whose return address is l (the regular predicate $\Gamma l \Gamma^*$ ensures that the return address of the procedure *proc* is l), the return value stored in the variable y should be eventually checked in all the future paths (i.e., *Test(y)*) inside this procedure (this is ensured by the fact that the stack is still of the form $\Gamma l \Gamma^*$ when the test of y is made). After this test, the socket y should be eventually closed in all the future paths (this is ensured by **EXAF** $close(y)$). The other rules in Table 1 are explained as follows.

The formula r_2 states that whenever *bind* is called to bind the socket to some address in a procedure whose return address is l, the user has to check whether the binding is correct before this procedure returns. r_3 and r_4 are similar to r_2.

The formula r_5 specifies that a socket y should be created ($y = socket(-,-,-)$) prior to binding the socket y to some address ($bind(y,-,-)$), where $-$ matches any constant (i.e., a variable quantified by $\forall$). r_6 is similar to r_5.

The formula r_7 states that any occurrence of $connect(y,-)$ should be preceded by an occurrence of $y = socket(-, SOCK\_STREAM, -)$ using the TCP Protocol.

Table 1. A set of API usage rules of the Socket Library extracted from the Socket library manual

No.	Rule
r_1	$\forall y\, \forall l\, \mathbf{AG}\left((y = socket(-,-,-) \wedge \Gamma l \Gamma^*) \Longrightarrow \mathbf{AF}\,(Test(y) \wedge \Gamma l \Gamma^* \wedge \mathbf{EX}\,\mathbf{AF}\, close(y))\right)$
r_2	$\forall y\, \forall l\, \mathbf{AG}\,(y = bind(-,-,-) \wedge \Gamma l \Gamma^* \Longrightarrow \mathbf{AF}\,(Test(y) \wedge \Gamma l \Gamma^*))$
r_3	$\forall y\, \forall l\, \mathbf{AG}\,(y = listen(-,-) \wedge \Gamma l \Gamma^* \Longrightarrow \mathbf{AF}\,(Test(y) \wedge \Gamma l \Gamma^*))$
r_4	$\forall y\, \forall l\, \mathbf{AG}\,(y = connect(-,-,-) \wedge \Gamma l \Gamma^* \Longrightarrow \mathbf{AF}\,(Test(y) \wedge \Gamma l \Gamma^*))$
r_5	$\forall y\, \mathbf{A}[y = socket(-,-,-)\, \mathbf{R}\, \neg bind(y,-,-)]$
r_6	$\forall y\, \mathbf{A}[listen(y,-)\, \mathbf{R}\, \neg accept(y,-,-)]$
r_7	$\forall y\, \mathbf{A}[y = socket(-, SOCK\_STREAM, -)\, \mathbf{R}\, \neg connect(y,-,-)]$
r_8	$\forall y\, \mathbf{A}[(y = socket(-, SOCK\_STREAM, -) \wedge \mathbf{A}[bind(y,-,-)\, \mathbf{R}\, \neg listen(y,-)])\, \mathbf{R}\, \neg listen(y,-)]$
r_9	$\forall y\, \mathbf{A}[connect(y,-,-) \vee y = accept(-,-,-)\, \mathbf{R}\, \neg send(y,-,-,-)]$
r_{10}	$\forall y\, \mathbf{A}[connect(y,-,-) \vee y = accept(-,-,-)\, \mathbf{R}\, \neg recv(y,-,-,-)]$
r_{11}	$\mathbf{AG}\, \forall y\, (y = accept(-,-,-) \Longrightarrow \mathbf{AF}\, close(y))$
r_{12}	$\forall y\, \mathbf{A}[y = socket(-, SOCK\_DGRAM, -)\, \mathbf{R}\, \neg(sendto(y,-,-,-,-,-) \vee recvfrom(y,-,-,-,-,-))]$
r_{13}	$\forall y\, \mathbf{A}[sendto(y,-,-,-,-,-) \vee bind(y,-,-)\, \mathbf{R}\, \neg recvfrom(y,-,-,-,-,-)]$

The formula r_8 specifies that any occurrence of listening to a socket y ($listen(y, -)$) should be preceded by an occurrence of creating the socket y using the TCP Protocol ($y = socket(-, SOCK\_STREAM, -)$), and the socket y should be bound to some address ($bind(y, -, -)$) before listening.

The formula r_9 states that before sending a data ($send(y, -, -, -)$) via a socket y, the socket y should either be connected to the target server at the client-side ($connect(y, -, -)$) or y should be the socket created by $y = accept(-, -, -)$ at the server-side. r_{10} is similar.

The formula r_{11} specifies that the new socket created by $y = accept(-, -, -)$ should be eventually closed ($close(y)$) in all the future paths.

The formula r_{12} states that the socket should be created using the UDP Protocol ($y = socket(-, SOCK\_DGRAM, -)$) prior to sending ($sendto(y, -, -, -, -)$) or receiving ($recvfrom\ (y, -, -, -, -)$) some data using the UDP Protocol.

The formula r_{13} specifies that before receiving ($recvfrom(y, -, -, -, -)$) some data using the UDP Protocol, one has to send some data ($sendto(y, -, -, -, -)$) to the server at the client-side or bind ($bind(y, -, -)$) the socket to some address at the server-side. Since using the UDP protocol, no connection is created, the client sends data by specifying the target address in the third parameter of the function *sendto*. After this, the client can receive data from the server. The server can send data only after receiving the client address from some client.

Checking the API Usage Rules. Consider the program in Figure 3. If *cond1* is true (Fig. 3: line 13), the socket s will never be closed. The SCTPL formula r_1 can detect this bug by model-checking the program against r_1. Consider the program in Figure 4, if the client managed to connect to a server which only supports the UDP Protocol as in Figure 5, the connection at line 5 of Figure 4 will fail, then sending (Figure 4: line 7) or receiving (Figure 4: line 9) some data via the socket s will induce an error. This error can be detected by checking the SCTPL formula r_4.

4 rSCTPL and The Procedure-Cutting Abstraction

To make API usage rules verification more efficient, it is important to model programs by PDSs having *small size*. We propose in this section to use the *procedure-cutting abstraction* to drastically reduce the size of the program model. The procedure-cutting abstraction removes all the procedures whose runs don't call any API function specified in the given SCTPL formula. We characterize a sub-logic rSCTPL of SCTPL that is sufficient to specify all the API usage rules that we met, and we show that the procedure-cutting abstraction preserves all rSCTPL formulas.

4.1 Procedure-Cutting Abstraction

Let $\mathcal{M}$ be a program that consists of a finite set of procedures $Proc = \{proc_i \mid 1 \leq i \leq m\}$. Each procedure $proc_i$ will generate transition rules in the PDS model. Imagine there exists some procedure $proc_j$ whose runs do not call any API function specified in the given SCTPL formula ψ, then removing $proc_j$ will not change the satisfiability of

ψ. This means that the procedure $proc_j$ can be cut. Cutting such procedure $proc_j$ will drastically reduce the size of the PDS model. We call this *procedure-cutting abstraction*. From the PDS's point of view, a function call statement $y = proc_j(...)$ at a control point n (suppose n' is the next control point of n) is represented by the transition rule $\rho = \langle s_0, n\rangle \hookrightarrow \langle s_0, e_{proc_j} n'\rangle$ where e_{proc_j} denotes the entry control point of the procedure $proc_j$. Whenever the procedure $proc_j$ can be cut, we will add the transition rule $\rho' = \langle s_0, n\rangle \hookrightarrow \langle s_0, n'\rangle$ instead of ρ. The transition rule ρ' expresses that the run from n will immediately move to n' without entering the procedure $proc_j$. By doing the procedure-cutting abstraction, the size of the stack alphabet and transition rules will be drastically reduced.

Formally, to compute the abstracted program, we proceed as follows. Let $\mathcal{M}$ be a program, a *call graph* of $\mathcal{M}$ is a tuple $G = (Proc, E, proc_0)$, where $Proc$ is a finite set of nodes denoting the procedure names of $\mathcal{M}$; $E \subseteq Proc \times Proc$ is a finite set of edges such that $(proc_i, proc_j) \in E$, denoted by $proc_i \longrightarrow proc_j$, iff $proc_j$ is called in the procedure $proc_i$; $proc_0 \in Proc$ is the initial node corresponding to the entry procedure (usually, the *main* function) of $\mathcal{M}$. A node $proc_i$ can reach the node $proc_j$ iff there exists a set of edges $proc_{k_1} \longrightarrow proc_{k_2}, ..., proc_{k_m} \longrightarrow proc_{k_{m+1}}$ in E such that $k_1 = i$ and $k_{m+1} = j$. Let $Op(\psi) = \{proc \in AP \mid \exists proc(x_1, ..., x_m) \in cl(\psi) \wedge proc \neq Test\}$ denote the set of atomic propositions (i.e., API function names) used in the SCTPL formula ψ except the additional atomic proposition *Test*. The procedure-cutting abstraction computes the abstracted program $\mathcal{M}'$ by (1) removing all the procedures $proc \in Proc$ s.t. the node $proc$ cannot reach any node of $Op(\psi)$ in G (i.e., the run of $proc$ will not call any function in $Op(\psi)$), and (2) replacing each function call $y = proc(p_1, ..., p_m)$ by a *skip* statement, i.e., no operation statement.

Proposition 1. *Given a program $\mathcal{M}$ and a SCTPL formula ψ, we can compute the abstracted program $\mathcal{M}'$ in linear time.*

4.2 The rSCTPL Logic

The procedure-cutting abstraction can drastically reduce the size of the program model. However, it cannot preserve all SCTPL formulas. Indeed, formulas using the **X** operator without any restriction are not preserved, since the procedure-cutting abstraction removes procedures in the programs and replaces some function calls by *skip*. However, formulas of the form $a(x_1, ..., x_m) \wedge \mathbf{EX}\phi$ and $a(x_1, ..., x_m) \wedge \mathbf{AX}\phi$ are preserved when ϕ is a regular predicate e or its negation $\neg e$ or a SCTPL formula using the **X** operator as in the above form. Indeed, if the predicate $a(x_1, ..., x_m)$ appearing in a SCTPL formula (a function call or a return value test) is made in some procedure $proc$, then all the procedures including $proc$ whose runs can reach $proc$ will not be removed by the procedure-cutting abstraction. This implies that the next control point of $a(x_1, ..., x_m)$ will not be removed and the stack content at the next control point in the abstracted program $\mathcal{M}'$ is the same as in $\mathcal{M}$.

Moreover, formulas using regular variable expressions (e.g. e, $\neg e$) without any restriction are not preserved. Indeed, control points in $\mathcal{M}$ satisfying e or $\neg e$ may be removed by the procedure-cutting abstraction. Thus, the runs of $\mathcal{M}'$ cannot reach these control points. However, formulas of the form $a(x_1, ..., x_m) \wedge e$ or $a(x_1, ..., x_m) \wedge \neg e$

are preserved. Since all the procedures which can reach the procedure *proc* where $a(x_1, ..., x_m)$ is made are not removed, each control point in $\mathcal{M}$ satisfying $a(x_1, ..., x_m)$ has the same calling procedures (i.e., stack content) as in $\mathcal{M}'$. Then, a configuration of $\mathcal{M}$ satisfies $a(x_1, ..., x_m) \wedge e$ iff this configuration of $\mathcal{M}'$ satisfies $a(x_1, ..., x_m) \wedge e$.

Based on the above observations, we define rSCTPL as follows (where $a(x_1, ..., x_m) \in AP_{\mathcal{X}}$, $x \in \mathcal{X}$, and $e \in \mathcal{R}$):

$$\begin{aligned}\varphi ::= & \ a(x_1, ..., x_m) \mid \neg a(x_1, ..., x_m) \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \forall x\, \varphi \mid \exists x\, \varphi \\ & \mid \mathbf{A}[\varphi \mathbf{U} \varphi] \mid \mathbf{E}[\varphi \mathbf{U} \varphi] \mid \mathbf{A}[\varphi \mathbf{R} \varphi] \mid \mathbf{E}[\varphi \mathbf{R} \varphi] \mid a(x_1, ..., x_m) \wedge \psi \\ \psi ::= & \ e \mid \neg e \mid \mathbf{EX} e \mid \mathbf{AX} e \mid \mathbf{EX} \neg e \mid \mathbf{AX} \neg e \mid \mathbf{EX} \varphi \mid \mathbf{AX} \varphi\end{aligned}$$

Intuitively, rSCTPL is a sub-logic of SCTPL, where (1) the next time operator **X** is used only to specify that a rSCTPL formula ψ or a regular predicate e or its negation $\neg e$ holds immediately after an atomic predicate holds (i.e., an API function call is made or a return value is tested), and (2) regular predicates and their negations are conjuncted with atomic predicates. rSCTPL is sufficient to specify all the API usage rules we met.

However, the procedure-cutting abstraction does not preserve rSCTPL formulas when a cut procedure has an infinite execution. For instance, let $n_1 \xrightarrow{stmt} n_2$ be an edge s.t. *stmt* is a function call $y = f(p_1, ..., p_m)$ and the procedure f has an infinite execution. Suppose we replace this function call by *skip*. If n_1 and all the control locations of f don't satisfy the atomic predicate a (i.e., API function calls or return value test), while n_2 satisfies a, then the configuration $\langle s_0, n_1\omega \rangle$ of $\mathcal{M}$ satisfies $\mathbf{EG}\neg a$, but $\langle s_0, n_1\omega \rangle$ does not satisfy $\mathbf{EG}\neg a$ in $\mathcal{M}'$ due to the removal of the infinite execution. On the other hand, if n_1 and all the control locations of f do not satisfy the atomic predicate a, while n_2 satisfies the atomic predicate b, then the configuration $\langle s_0, n_1\omega \rangle$ of $\mathcal{M}'$ satisfies $\mathbf{A}[\neg a \mathbf{U} b]$ due to the removal of the infinite execution, while $\langle s_0, n_1\omega \rangle$ does not satisfy $\mathbf{A}[\neg a \mathbf{U} b]$ in $\mathcal{M}$ (since b is never true in the infinite execution). We can show the following theorem.

Theorem 2. *Let ψ be a rSCTPL formula. Let $\mathcal{M}$ be a program and $\mathcal{M}'$ be the program obtained from $\mathcal{M}$ by applying the procedure-cutting abstraction. Let $\mathcal{P}$ (resp. $\mathcal{P}'$) be the PDS modeling the program $\mathcal{M}$ (resp. $\mathcal{M}'$). If all the removed procedures are infinite execution free, then $\mathcal{P}$ satisfies ψ iff $\mathcal{P}'$ satisfies ψ.*

5 Experiments

We implemented our techniques in a tool for API usage rules verification. Given a program $\mathcal{M}$ using some libraries which are equipped with the API usage rules specified in rSCTPL, our tool automatically answers either Yes or No, depending on whether the program violates the API usage rules or not.

In our implementation, we use goto-cc [31] to parse ANSI-C programs into goto-cc binary programs. We implemented a translator translating goto-cc binary programs into pushdown systems and outputs the required predicates as discussed in Section 3.3. We use the SCTPL model-checker of [41] as engine. In our experiments, we consider several API usage rules: the socket library API usage rules and the file operation usage rules. We checked several open-source C programs against these API usage rules. All the experiments were run on a Linux platform (Fedora 13) with a 2.4GHz CPU and 2GB

of memory. The time limit is fixed to 30 minutes. Our tool detected several previously unknown errors in some well-known open source programs. The run time consists of the time spent for parsing goto-cc binary programs and model-checking. It excludes the time for translating ANSI-C programs into goto-cc binary programs. We also run our tool without considering the procedure-cutting abstraction. We observed that the procedure-cutting abstraction significantly speeds up the analysis.

5.1 Checking The Socket Library API Usage Rules

To check the socket library API usage rules shown in Table 1, we checked seven open-source programs from SourceForge [12] which are written in C and use the socket library, and four generic tutorial socket programs written by Seshadri [37].

The benchmark contains the following programs. **Comserial** is a program that helps turn console application into a web based service, by reading from TCP connections and providing commands from each connection to applications through a socket. **MrChaTTY** is a chat program that allows users to chat via UNIX terminals through sockets. **Mrhttpd** is a web server. **Nerv** is a common socket server. **Nssl** is a netcat-like program with SSL support. **Pop3client** is a mail client which reads mail in a console and connects to servers using POP3 Protocol. **Ser2nets** is a program allowing network connections to remote serial ports. **TCPC**, **TCPS**, **UDPC** and **UDPS** are a TCP client, a TCP server, a UDP client and a UDP server tutorial programs, respectively.

Table 2 shows the results of checking the socket library API usage rules with the procedure-cutting abstraction. The row #*LOC* gives the number of lines of the program. For $1 \leq i \leq 13$, the row r_i depicts the results of checking the API usage rule r_i against these programs, where the rows $Time(s)$ and $Mem(MB)$ give the time consumption in seconds and memory consumption in MB, respectively. The result *Proved* denotes that the program satisfies the corresponding API usage rule, *FA* denotes *false alarm* and *Bug* denotes a real bug. *o.o.m.* (resp. *o.o.t.*) means run out of memory (resp. time).

We can see from Table 2, there are 22 alarms including *Bug* and *FA*. We found that 12 of these alarms are real bugs and the others are false alarms. These false alarms arose from the fact that we abstract away the data. We found 12 real errors in these programs. For instance, the program **Comserial** does not call *listen* before calling *accept* in the file *passwdserver.c* when *argc* is 1. Moveover, most of these programs will not close the socket by calling *close* nor check the return values of *socket* in some paths. E.g., **Comserial** does not check the return value (i.e., socket) in the file *comserver.c* before it is used. In the file *main.c*, when it fails in binding a socket to some address, **Mrhttpd** will not close this socket before the program terminates.

5.2 Checking File Operation Usage Rules

File reading and writing are frequently used in programs. To read or write a file, a user has to correctly open the file by calling *fopen* which returns a file pointer to the file. Then the user can read from or write to that file. Finally the file pointer should be closed by calling *fclose*.

For file operation API usage rules, we consider two rules from *stdio.h*: $F_1 = \mathbf{AG} \forall y \Big(y = fopen(-,-) \Longrightarrow \mathbf{AF}(Test(y) \wedge \mathbf{EXAF} fclose(y))\Big)$ and $F_2 = \forall y \, \mathbf{A}[y = fopen(-,-)$

Table 2. Results of checking the socket library API usage rules with the procedure-cutting abstraction

Program		Comserial	MrChaTTY	Mrhttpd	Nerv	Nssl	Pop3client	Ser2nets	TCPC	TCPS	UDPC	UDPS
#LOC		1.0k	1.2k	1.4k	1.1k	1.1k	1.6k	7.3k	70	90	50	60
r_1	Time(s)	0.08	0.26	0.29	7.94	1.24	0.41	70.53	0.01	0.01	0.01	0.01
	Mem(MB)	0.24	0.44	0.66	5.94	1.44	0.58	11.63	0.09	0.13	0.06	0.06
	Result	Bug	FA	Bug	FA	Bug	Bug	Bug	Bug	Bug	Bug	Bug
r_2	Time(s)	0.01	0.09	0.01	0.04	0.23	0.01	8.72	0.01	0.01	0.01	0.01
	Mem(MB)	0.06	0.35	0.07	0.24	0.36	0.01	2.04	0.01	0.01	0.01	0.01
	Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
r_3	Time(s)	0.01	0.09	0.01	0.03	0.11	0.01	9.57	0.01	0.02	0.01	0.01
	Mem(MB)	0.05	0.37	0.07	0.20	0.29	0.01	2.03	0.01	0.10	0.01	0.01
	Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
r_4	Time(s)	0.01	0.01	0.01	0.11	0.16	0.09	6.31	0.01	0.01	0.01	0.01
	Mem(MB)	0.01	0.01	0.01	0.29	0.33	0.29	1.72	0.07	0.01	0.01	0.01
	Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
r_5	Time(s)	0.01	0.01	0.01	0.01	0.01	0.01	0.18	0.01	0.01	0.01	0.01
	Mem(MB)	0.04	0.18	0.05	0.22	0.19	0.14	1.07	0.04	0.06	0.04	0.04
	Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
r_6	Time(s)	0.06	0.01	0.01	0.01	0.01	0.01	0.20	0.01	0.03	0.01	0.01
	Mem(MB)	0.15	0.18	0.05	0.19	0.01	0.01	1.12	0.01	0.10	0.01	0.01
	Result	Bug	Proved	Proved	Proved	FA	Proved	Proved	Proved	Proved	Proved	Proved
r_7	Time(s)	0.01	0.01	0.01	0.02	0.02	0.02	0.21	0.01	0.01	0.01	0.01
	Mem(MB)	0.04	0.15	0.05	0.22	0.19	0.18	0.92	0.05	0.05	0.04	0.04
	Result	Proved	Proved	Proved	Proved	Bug	FA	Proved	Proved	Proved	Proved	Proved
r_8	Time(s)	0.01	0.07	0.01	0.09	0.07	0.03	1.03	0.01	0.01	0.01	0.01
	Mem(MB)	0.07	0.47	0.08	0.54	0.44	0.30	2.86	0.07	0.12	0.05	0.05
	Result	Proved	Proved	Proved	Proved	FA	Proved	Proved	Proved	Proved	Proved	Proved
r_9	Time(s)	0.01	0.01	0.01	0.02	0.01	0.01	0.07	0.02	0.01	0.01	0.01
	Mem(MB)	0.11	0.34	0.30	0.50	0.29	0.30	1.46	0.08	0.10	0.01	0.01
	Result	Proved	FA	Proved	Proved	Proved	FA	Proved	Proved	Proved	Proved	Proved
r_{10}	Time(s)	0.01	0.01	0.01	0.05	0.01	0.01	0.07	0.01	0.01	0.01	0.01
	Mem(MB)	0.11	0.33	0.33	0.75	0.29	0.35	1.46	0.08	0.09	0.01	0.01
	Result	Proved	FA	Proved	FA	Proved	FA	Proved	Proved	Proved	Proved	Proved
r_{11}	Time(s)	0.10	0.56	0.32	-	-	0.13	-	0.02	0.03	0.01	0.01
	Mem(MB)	0.47	1.97	1.50	o.o.m.	o.o.m.	0.39	o.o.m.	0.11	0.17	0.01	0.01
	Result	Bug	Proved	Proved	-	-	Proved	-	Proved	Proved	Proved	Proved
r_{12}	Time(s)	0.01	0.01	0.01	0.01	0.01	0.01	0.04	0.01	0.01	0.01	0.01
	Mem(MB)	0.04	0.15	0.05	0.18	0.15	0.14	0.71	0.04	0.05	0.05	0.04
	Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
r_{13}	Time(s)	0.01	0.01	0.01	0.01	0.01	0.01	0.07	0.01	0.03	0.01	0.01
	Mem(MB)	0.05	0.31	0.07	0.17	0.30	0.01	1.46	0.01	0.10	0.05	0.05
	Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved

$\mathbf{R}\neg(fread(-,-,-,y) \vee fwrite(-,-,-,y))]$. F_1 states that whenever a file is opened by calling *fopen* where y stores its return file pointer (i.e., $y = fopen(-,-)$), we need to check whether the opening of the file is correct (i.e., *Test(y)*), and there exists a next point after checking y such that the file is eventually closed (i.e., **EX AF** $fclose(y)$). F_2 states that the user cannot read from or write to a file pointer y unless the file pointer y points to some file (i.e., has already been opened).

To evaluate these two rules, we checked the following open source programs which use file API functions from *stdio.h*. **Verbs** is a bounded model checker [10]. **Getafix** is a symbolic model-checker for recursive boolean programs [3]. **Moped** is a model-checker for pushdown systems [7]. **Acacia+** is a tool for LTL realizability and synthesis [1]. **Mist** is a solver of the coverability problem for monotonic extensions of Petri nets [6]. **Elastic** is a translator from elastic specifications to hytech or uppaal language [2]. **Mckit**

Table 3. Results of checking the API usage rules F_1 and F_2 with the procedure-cutting abstraction

Program		Verbs	Getafix	Moped	Acacia+	Mist	Elastic	Mckit	TSPASS	MiniSat	Walksat	Ubcsat
#LOC		4.0k	11.5k	30.3k	8.0k	16.0k	15.4k	26.7k	62.3k	1.4k	1.4k	16.9k
F_1	Time(s)	0.96	0.18	9.92	0.05	0.01	1.45	-	0.25	0.01	0.06	216.88
	Mem(MB)	1.17	0.36	10.52	0.20	0.10	2.99	o.o.m.	0.67	0.08	0.28	15.92
	Result	Bug	Bug	Proved	Bug	Proved	Proved	-	Proved	Proved	Bug	FA
F_2	Time(s)	0.08	0.29	9.67	0.01	0.26	0.89	23.60	0.01	0.01	0.01	0.06
	Mem(MB)	0.50	0.84	10.26	0.09	0.90	2.94	15.00	0.27	0.27	0.13	0.89
	Result	FA	Proved	FA	Proved	FA	FA	Proved	Proved	FA	Proved	Proved

is a model-checking Kit [4]. **TSPASS** is a fair automated theorem prover for monodic first-order temporal logic with expanding domain semantics and propositional linear-time temporal logic [8]. **Walksat**, **MiniSat** and **Ubcsat** are three SAT solvers [5, 9, 11].

Table 3 shows the results of checking these programs against F_1 and F_2 with the procedure-cutting abstraction. As shown in Table 3, we found that **Verbs**, **Getafix**, **Acacia+** and **MiniSat** have real errors. E.g., in the file *main.c*, **Verbs** does not close an opened file by calling *fclose* before the program terminates. Moreover, in the files *issat.c*, *main.c* and *util.c*, a file pointer is used without checking whether it is *NULL* or not (i.e., whether the file exists or not). **Acacia+**, **Walksat** and **Getafix** do not close opened files which are opened in *main.c*, *walksat.c*, *bpsuspend.y* and *bp.y*, respectively.

6 Related Work

There has been a lot of works on API usage rules specification and checking [13–16, 19, 22, 24–26, 28, 30, 32–36, 38, 44–47]. However, all these works cannot specify context-sensitive specifications, whereas our approach can.

Some tools dedicated to software model-checking were used to check API usage rules for device drivers, such as DDVerify [46]. But, these tools can only check safety properties. Other works on software model-checking, such as [17, 18, 27, 42, 43], could be applied to check API usage rules. However, all these works cannot check full CTL properties.

Model-checking is used to verify security-critical applications in which security vulnerabilities are expressed by safety properties over API functions [20, 21]. However, these works consider only safety properties.

Code contracts introduced in [24] can specify pre/post-conditions and invariants for each API function. Programmers have to make sure that a pre-condition (resp. post-condition) holds at the entry (resp. exit) of each API function, and that invariants always hold inside the API function. These code contracts can be verified via either runtime checking or static checking at compile time. However, they cannot specify relations between API functions which are often used in API usage rules.

Mining-based methods are proposed [13–15, 19, 22, 25, 26, 30, 32, 33, 35, 38, 45, 47] to discover API usage rules from executing traces or source codes, where API usage rules are represented by some patterns or finite automata. One can apply model-checking techniques to check whether programs violate or not API usage rules represented by patterns or finite automata. However, all these works cannot specify data dependencies

between API functions' parameters and return values of API functions. This disallows one to precisely express API usage rules. Variables are introduced into finite automata to specify data dependencies between API functions in [15, 28]. However, these works cannot express CTL-like properties (e.g., the above file operation API usage rule), and do not show how to check whether programs violate or not API usage rules represented by finite automata equipped with variables.

A class of temporal properties, called QBEC, is used to specify API usage rules using at most one temporal operator [34]. We can show that SCTPL is more expressive than QBEC. Indeed, all the temporal operators in QBEC can be expressed by SCTPL formulas. Ramanathan et al propose a formalism in [36] to specify data-dependence between API functions. However, they only consider mining preconditions of API functions rather than verification. CTL extended with variables is proposed to specify API usage rules in [44]. However, this work cannot specify context-sensitive specifications which is important for API usage rules.

SCTPL is introduced in our previous work [41], in which SCTPL is used to express malicious behaviors and model-checking is applied to detect malware. Although, SCTPL is as expressive as CTL with regular valuations [39], in [41], we have shown that SCTPL model-checking is more efficient than CTL model-checking with regular valuations.

References

1. Acacia+, http://lit2.ulb.ac.be/acaciaplus/
2. elastic, http://www.ulb.ac.be/di/ssd/madewulf/aasap/
3. Getafix, http://www.cs.uiuc.edu/madhu/getafix/
4. Mckit, http://www.fmi.uni-stuttgart.de/szs/tools/mckit/
5. Minisat, C.: language version, http://minisat.se/MiniSat.html
6. Mist2, http://software.imdea.org/pierreganty/software.html
7. Moped, http://www.fmi.uni-stuttgart.de/szs/tools/moped/
8. Tspass, http://www.csc.liv.ac.uk/michel/software/tspass/
9. Ubcsat, http://ubcsat.dtompkins.com/
10. Verbs, http://lcs.ios.ac.cn/zwh/verbs/index.html
11. Walksat, version 35, http://www.cs.rochester.edu/kautz/walksat/
12. SourceForge (2012), http://sourceforge.net
13. Acharya, M., Xie, T.: Mining API error-handling specifications from source code. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 370–384. Springer, Heidelberg (2009)
14. Acharya, M., Xie, T., Pei, J., Xu, J.: Mining API patterns as partial orders from source code: From usage scenarios to specifications. In: ESEC/FSE 2007 (2007)
15. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: POPL (2002)
16. Besson, F., Jensen, T.P., Métayer, D.L.: Model checking security properties of control flow graphs. Journal of Computer Security (2001)
17. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. In: STTT (2007)
18. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Trans. Software Eng. 30(6) (2004)
19. Chen, F., Roşu, G.: Mining parametric state-based specifications from executions. Technical report (2008)

20. Chen, H., Dean, D., Wagner, D.: Model checking one million lines of C code. In: NDSS (2004)
21. Chen, H., Wagner, D.: Mops: an infrastructure for examining security properties of software. In: ACM Conference on Computer and Communications Security (2002)
22. Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining object behavior with ADABU. In: WODA (2006)
23. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithm for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, Springer, Heidelberg (2000)
24. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 10–30. Springer, Heidelberg (2011)
25. Gabel, M., Su, Z.: Javert: fully automatic mining of general temporal properties from dynamic traces. In: FSE (2008)
26. Gabel, M., Su, Z.: Symbolic mining of temporal specifications. In: ICSE (2008)
27. Godefroid, P.: Software model checking: The Verisoft approach. Formal Methods in System Design 26 (2005)
28. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. In: ESEC/SIGSOFT FSE (2005)
29. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. In: Julisch, K., Kruegel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 174–187. Springer, Heidelberg (2005)
30. Kremenek, T., Twohey, P., Back, G., Ng, A.Y., Engler, D.R.: From uncertainty to belief: Inferring the specification within. In: OSDI (2006)
31. Kroening, D.: CBMC (2012), `http://www.cprover.org/cbmc`
32. Liu, C., Ye, E., Richardson, D.J.: Software library usage pattern extraction using a software model checker. In: ASE (2006)
33. Lo, D., Khoo, S.-C.: SMArTIC: towards building an accurate, robust and scalable specification miner. In: FSE 2006 (2006)
34. Lo, D., Ramalingam, G., Ranganath, V.P., Vaswani, K.: Mining quantified temporal rules: Formalism, algorithms, and evaluation. In: WCRE (2009)
35. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: ICSE 2008 (2008)
36. Ramanathan, M.K., Grama, A., Jagannathan, S.: Static specification inference using predicate mining. In: PLDI (2007)
37. Seshadri, P.: Generic Socket Programming tutorial (2008), `http://www.prasannatech.net/2008/07/socket-programming-tutorial.html`
38. Shoham, S., Yahav, E., Fink, S.J., Pistoia, M.: Static specification mining using automata-based abstractions. IEEE Trans. Software Eng. (2008)
39. Song, F., Touili, T.: Efficient CTL model-checking for pushdown systems. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 434–449. Springer, Heidelberg (2011)
40. Song, F., Touili, T.: Efficient malware detection using model-checking. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 418–433. Springer, Heidelberg (2012)
41. Song, F., Touili, T.: Pushdown model checking for malware detection. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 110–125. Springer, Heidelberg (2012)
42. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. (2003)
43. Visser, W., Mehlitz, P.C.: Model checking programs with java pathFinder. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 27–27. Springer, Heidelberg (2005)

44. Wasylkowski, A., Zeller, A.: Mining temporal specifications from object usage. Autom. Softw. Eng. (2011)
45. Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: ESEC/FSE (2007)
46. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent linux device drivers. In: ASE (2007)
47. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: ICSE (2006)