Model-Checking for Software Vulnerabilities Detection With Multi-Language Support

Rachid Hadjidj, Xiaochun Yang, Syrine Tlili, Mourad Debbabi
Computer Security Laboratory
Concordia Institute for Information System Security
Concordia University, Montreal, Quebec, Canada
hadjidj,xc_yang,s_tlili,debbabi@encs.concordia.ca

Abstract

In this paper we develop a security verification framework for open source software with a multi-language support. We base our approach on the GCC compiler which is considered as the defacto open source compiler for several languages including C, C++, JAVA, ADA, FORTRAN, etc. To achieve our goal we use a conventional push down system model-checker for reachability properties, and turn it into a fully fledged verification tool for both low and high level software security properties. We also allow programmers to define a wide range of temporal security properties using an automata-based specification approach. As a result, our approach can model-check large scale software against system-specific security properties.

1 Introduction

In software systems, programming errors are usually the root cause leading to security breaches such as denial of service, buffer overflow, format string, code injection, etc. Coding errors may result either from defectively designed language features such as no built-in protection for accessing memory in C, or from invalid logic having high-level semantic error. At the same time, the growing size and complexity of software systems render the task of errors detection more and more difficult. In order to produce correct software, many efforts have been put into developing formal methods for automated program verification. Both static analysis and model checking are used to provide tools for the automated detection of programming errors in source code.

Static analysis operates at compile-time on an intermediate representations of the source code generated by the compiler, and has been mainly used for code optimization such as eliminating dead code, detecting infinite loops, uninitialized variables, unused variables, etc. Nowadays, static analysis is also considered as a fundamental block for security and safety verification of software [13, 12, 7]. This technique takes advantage of control flow, data flow, and type information generated by the compiler to predict undesirable behavior of the program at early stages of software life cycle. However, static approaches can only detect security properties that syntactically matches concrete program actions. Therefore, they are not suitable for verifying high-level properties related to the program security functionalities.

New trends in software model-checking show great promise in detecting programming errors and ensuring the correctness of software [10, 1, 4]. Model-checking is more efficient than static analysis in specifying and detecting system-specific security properties. However, the main issue with this approach is to generate a concise model-checkable abstraction of the program. The abstraction should be precise enough in order not to over-estimate nor to under-estimate the program behavior. In fact, the precision of the analysis relies on the expressiveness of the program abstraction. The effort and the time required for the programmer to build a program model makes model-checking less suitable than static analysis for large program verification unless this process is automated. In this paper, we define a security verification framework that brings static analysis and model-checking into a synergy in order to leverage the advantages and overcome the shortcomings of both techniques. The core idea is to utilize static analysis in order to automatically build a model-checkable abstraction of the program to verify. We also allow programmers to define a wide range of temporal security properties using an automata-based specification approach. As a result,
our approach can model-check large scale software against system-specific security properties.

To achieve our goal, we use a conventional push down system model-checker called Moped [17, 11]. Moped takes as an input a program model written in a procedural language called Remopla, which is then translated into an equivalent pushdown system representation. Moped allows to verify simple reachability properties by looking for the reachability of a specific statement in the Remopla code. Though interesting, this capability is not directly sufficient for verifying security properties. In fact, a security properly is the description of a pathological behavior in the execution of program. Such a behavior requires is general an elaborate formalism to be specified and can rarely be stated as the simple reachability of a specific statement in the program. To specify security properties we use the formalism of security automata. A security automaton is a simple automaton with two spacial states: start and error, and where transitions map to instructions or statement in the program to verify. The reachability of the error state in the security automaton when synchronized with the program’s behavior is an indication of the occurrence of the pathology. To overcome the limitation of Moped in this regard, we translate a security automaton into a Remopla representation then synchronize it with the Remopla model of the program to verify. This comes to synchronizing the push down systems of the program and the security automaton. The problem of verifying a security property is translated to detecting the reachability of the error state in the synchronized model.

Our ultimate goal in this work was to develop a security verification tool for open source software with a multi language support. Hence, we based our approach on the GCC compiler which is considered as the de facto open source compiler for several languages including C, C++, JAVA, ADA, FORTRAN, etc. Recently, the GCC mainline includes the TREE-SSA framework [14] that facilitates static analysis with its GIMPLE intermediate representation of source code common to all supported languages. This provides easy access to flow and type information that are relevant for security verification. We abstract the required knowledge from the GIMPLE representation and automatically map it to Remopla representation to achieve multi-language support. Security properties and Remopla model are input to our security verification tool in order to detect and provide witness paths of detected security violations.

The remainder of this paper is organized as follows: The software components and the overall architecture of our tool are informally outlined in Section 2. Section 3 is dedicated to the generation of Remopla models of security properties and program source code. We present in Section 4 the implementation and the experimental results of our security verification tool. We discuss related work in Section 5 and conclude this paper in Section 6.

2 Approach Overview

In this section, we give an informal overview of our security verification environment. We first give a short introduction of the software components that provide the basis for our implementation, then, we describe the architecture depicted in Figure 1 that integrates these components and the modus operandi of our approach.

![Figure 1. Overall architecture of our security verification environment.](image)

2.1 Moped model-checker

Moped is a model-checking tool for pushdown systems (PDS) based on the algorithms defined in [6]. There are currently two versions of Moped. The first version performs a combined linear temporal logic and reachability model-checking on pushdown systems. Since 2005, the second version of Moped only performs reachability analysis, but comes with a more expressive modeling formalism called Remopla. Remopla is a Promela-like language that allows to describe systems using a procedural language which allows for recursive calls. Our software verification environment is based on the new version of Moped in order to benefit from the expressiveness of the Remopla language.

2.2 Tree-SSA Framework

Our ultimate goal was to provide a security verification environment for open source software. To achieve our objective, the GCC compiler fulfilled our requirement for the multi-language support. Since the last decades, the GCC compiler is considered as the de facto compiler for open

Authorized licensed use limited to: CONCORDIA UNIVERSITY LIBRARIES. Downloaded on September 30, 2009 at 17:14 from IEEE Xplore. Restrictions apply.
source software. Moreover, starting from version 4.0, the GCC mainline includes the TREE-SSA framework used for code optimization and static analysis. It provides the language and platform independent GIMPLE tree representation of source code that facilitates the analysis of the compiler intermediate code. At GIMPLE level, GCC linearizes all the high-level control flow structures including nested functions, exception handling, and loops, and breaks expressions down into a 3-address form, using temporary variables to hold intermediate values. Working with GIMPLE representation allows our analysis to be focused more on data modifications and control flow information instead of putting effort into analyzing complex language constructs.

2.3 Architecture

Figure 1 shows the architecture of our security verification environment. It consists of several phases like: property specification, program model extraction, and property model-checking. In the following paragraphs, we describe the input, output, and task of each one of these phases.

Phase1: Security Property Specification.

- Input: Security Policy.
- Output: Remopla model of security properties.

The first step of our security verification process requires the definition of a security policy that describes the program vulnerabilities in terms of security properties. We provide programmers with a tool in order to graphically characterize security properties that the program should verify. Each property is represented with a finite state automaton where the nodes define program states and the transitions match program actions. Such an automaton is called security automaton. To ease the property specification, our tool supports syntactical pattern matching for program expressions. For instance, the programmer can define an automata transition with the pattern `memAllocate()` that matches the `malloc` family of functions in C code. The pattern `free(X)` matches the deallocation operation of any pointer variable denoted with the variable pattern X. Finally, our tool automatically serializes the graphically specified properties into a Remopla specification that will be used as part of the input to the Moped model-checker.

Phase2: Program Model Extraction.

- Input: Program source code and specified security properties.
- Output: Remopla model that synchronizes the Remopla model of the program and the Remopla model of security properties.

The model extraction exploits an extended version of the GCC compiler (GCC 4.2.0) to parse source code of the program to verify, and dump its GIMPLE representation into XML files, which are then translated to a Remopla model: the program’s model. Given the previously specified security properties, our tool first converts them to a Remopla representation, then combines them with the generated program’s model. The combination of the security properties with the program’s model is performed in order to: (1) synchronize the behavior of the program and the one of the security automata, (2) resolve the patterns of security automata and match them with actual values taken from the source code, and (3) reduce the size of the program’s model by only considering program actions that are relevant to the specified security properties. The resulting Remopla model is then given as input to the Moped model-checker for reachability analysis.

Phase3: Property Model-Checking.

- Input: Remopla model.
- Output: Detected errors with execution traces.

The model-checking is the ultimate step of our process. The generated Remopla model is given as input to the Moped model-checker. An error is reported when a security automaton specified in the model, reaches an error state. Note that the published version of Moped has a shortcoming in the sense that it stops the verification at the first encountered error state. In order to allow detecting more than one error, we extend Moped with the capability to resume the verification after each error detection and report all possible errors. Moreover, we developed an error trace generation functionality in order to provide the programmer with witness paths of security violations.

3 Constructing Formal Models

This section presents the process of generating the Remopla representation of security properties and the the Remopla model of the program to analyze.

3.1 Modeling Security Properties

We define security properties as finite state automata where nodes represent states of the program’s model, and labeled transitions represent security relevant program operations. An implicit transition from a node back to itself is taken if no outgoing transitions match the given program

1An XML file is generated for each source file.
2This phase corresponds to an abstraction of the program’s model.
operation. Sequences of operations accepted by security automata represent execution paths violating security properties in question.

Our property specification approach supports syntactical pattern matching for variables and program statements. Consider the property automaton of Figure 2. It states that the validity of user-space pointer should be checked before it is accessed. In the property automaton, the transition `Check_Access(X)` is followed if the current program action matches the system call `verify_area(*)`, `X, *` or `access_ok(*, X, *)`. Then, the transition `Access_Userspace(X)` is matched when the system call `get_user(*, X)`, `put_user(*, X)`, `copy_from_user(*, X, *)`, `copy_to_user(X, *, *)`, or `__copy_from_user_inatomic(*, X, *)` is encountered. Moreover, the pattern `X` should syntactically match the same pointer variable in both transitions. We use the asterisk to denote parameters that are not significant for the verification. During program model generation, we substitute pattern variables of security automata with actual values from the source code in order to derive instances of the generic security automata specified by the user.

### 3.2 From Security Automata to Remopla

The second step of property specification translates a specified security automaton into a Remopla representation we also call Remopla automaton. We benefit from the expressiveness of the Remopla language to specify security automata as Remopla modules\(^3\). The Remopla module of Figure 3 illustrates the Remopla translation of the security automaton in Figure 2. The nodes and the transition labels of a security automaton are mapped to Remopla constructs, as defined hereafter:

- The automaton nodes are defined as elements of a Remopla enumeration type. In the example of Figure 3, the enumeration variable `states` is introduced to represent nodes of the security automaton. Each state has a unique integer value. We define the state values `start` and `error` to represent the automaton initial state and final state, respectively. Since the security automaton actually defines the negation of a security property, the final state is the risky state. The integer variable `current_state` is used to track the current state of the security automaton. The predefined label `INITIALIZATION` allows to define initialization operations. We initialize the variable `current_state` to the automaton initial state, i.e., the value `start`.

- The automaton alphabet defines the transition labels that denote security relevant program actions. Each label is represented by a Remopla constant as defined in Table 1. The first two entries define respectively func-

\(^3\)A Remopla module is similar to a procedure in a procedural language.
tion calls without arguments and function calls with arguments. Each function argument is an element of a global Remopla array `ARG[]` which is inquired during the model-checking process when function parameters are involved in the property verification. The third entry is the representation of an assignment operation. This mapping focuses on the program action, but not on the assigned data value. The next two entries denote respectively the entry point of function and its exit points. The pattern `f` will match the actual function name relevant to the security property in question. In the Remopla automaton of Figure 3, the symbol `f` matches system calls `access_ok()`, `verify_access()`, `get_user()`, etc. The last element represents the termination of program execution.

Now that we have defined the Remopla constructs for nodes and transition labels, we are able to map each security automata to a Remopla module. The latter defines the initial node and the destination node of each transition of a security automaton. We establish a synchronization between the Remopla model of the program and the Remopla module of properties. By synchronization, we mean that a property module transits from a state to another, if the outgoing transition label from the current state matches the encountered program action. A security violation is detected if one of the executed Remopla modules reaches the final state that is considered as risky state.

### 3.3 Program Model Extraction

The model extraction is the process that translates program source code to Remopla representation. The translation comprises two steps: The first one consists in dumping The GIMPLE representation of parsed source code into XML files, the second one translates the GIMPLE representation into a Remopla model.

```c
int main() {
  int error;
  void* kernel_ptr;
  int* user_ptr1, *user_ptr2;
  if (access_ok(VERIFY_READ, user_ptr1, sizeof(int)))
    copy_from_user(kernel_ptr, user_ptr1, sizeof(int));
  copy_to_user(user_ptr2, kernel_ptr, sizeof(int));
  return 0;
}
```

Figure 4. Sample source code to illustrate Remopla model generation.

```c
enum actions {
  ACTION_FUNCTION_CALL_copy_to_user,
  FUNCTION_CALL_copy_from_user,
  ACTION_FUNCTION_CALL_access_ok,
  ACTION_PROGRAM_END};

enum args {ARG_4, ARG_user_ptr2, ARG_0, ARG_user_ptr1, ARG_kernel_ptr};

int ARG[10];
int current_state;

module void copy_from_user (){
  move_state(ACTION_FUNCTION_CALL_copy_from_user);
  return;
}

module void copy_to_user (){
  move_state(ACTION_FUNCTION_CALL_copy_to_user);
  return;
}

module void access_ok (){
  move_state(ACTION_FUNCTION_CALL_access_ok);
  return;
}

module void main (){ ARG[0]=ARG_0; ARG[1]=ARG_user_ptr1; ARG[2]=ARG_4;
  access_ok();
  if (true ->
    ARG[0]=ARG_kernel_ptr; ARG[1]=ARG_user_ptr1; ARG[2]=ARG_4;
    copy_from_user();
  else -> break;
  fi;
  ARG[0]=ARG_user_ptr2; ARG[1]=ARG_kernel_ptr; ARG[2]=ARG_4;
  copy_to_user();
  return;
}
```

Figure 5. Remopla model of the source code in Figure 4.

As a developing technology for information representation, XML is suitable for representing GIMPLE trees for its effectiveness of documenting structured information. Decoupling our analysis tool from the GCC compiler is another reason for dumping GIMPLE trees to XML files. In our framework, the translator from GIMPLE trees to XML representation is the only component that needs to be maintained as GCC evolves. Hence, this approach speeds up the development of our analysis tool, obviating the complexity of programming inside a compiler.

We use Remopla constructs defined in Table 1 in order to map XML representation of program actions to Remopla. As such, we are able to match Remopla constructs in the program model with the Remopla constructs representing the security automata transitions. With this matching, the model-checker parses the program model and changes the automata states simultaneously. A state transition is performed when its label matches the encountered Remopla constructs in the model.
For example, Figure 5 shows the Remopla program for the sample code in Figure 4. For clarity, Figure 5 shows only statements relevant to our discussion. The variables `actions` and `args` contain all the program actions and passed parameters, respectively. The program model can be in one of the states defined in a property automaton. For this example, we consider the memory access property of Figure 3. The initial state of the program corresponds to the initial state of the considered security automaton.

For each security relevant actions in the source code, we perform the following:

- The global Remopla array `ARG[]` for function arguments is initialized to the list of arguments of the considered action. In the `main` module, before calling the function `access.ok()`, we store all its arguments in the global array `ARG[]`. We update the values of the Remopla array `ARG[]` when we encounter other function calls in the Remopla model.

- A Remopla module of the function call is generated in order to synchronize the program state with that of the security automaton. The synchronization is done by calling the `move_state` module of the security property with the program action given as a parameter.

The control flow skeleton of the source code is also extracted during the Remopla model generation. However, the control-flow analysis in our tool is path-insensitive at this time. We consider all paths in the source code without pruning infeasible paths (false positives).

The model extraction process discussed above produces Remopla models with large number of function parameters and program actions. As shown in Figure 5, both actions and parameters are represented as integer enumeration type. Since Moped is not always efficient when dealing with large range integer, the solution is to take into account the specified security properties during model extraction. Program actions are divided into two categories: property-relevant actions and property-irrelevant actions. Given a set of properties, most program statements are irrelevant to the security verification. These irrelevant actions are mapped to a default Remopla construct `ACTION_FUNCTION_CALL`, where the name of the function is omitted. Hence, we are able to reduce the size of the Remopla model and enhance the scalability of our security verification tool.

### 4 Implementation and Experiment Results

The experiment described in this section serves the purpose in demonstrating the usability and scalability of our tool. Our analysis tool is written in Java and uses Moped as the model-checker. We experiment our tool on real-world open source projects with a set of security properties. The hardware platform used to perform the experiments is a Dell D810 with Pentium M 1.86GHz CPU and 500M memory that runs Fedora Core 8.

We focus on temporal properties, which dictate the order in which security-relevant operations take place. At the current stage, we have defined a set of security properties that are specific to the C programming language, such as time-of-check-to-time-of-use (TOCTOU), null-pointer dereferences, memory leaks, and bad usage of temporary files. Properties are graphically modeled as finite automata. As an an example, Figure 6 shows the automaton specifying the TOCTOU, where we match the transition patterns with 49 function calls in the C language [2]. This property describes race condition attacks, which result from the fact that a sequence of system calls is not executed atomically. A common example is a process that checks access permission of an entity (e.g. a file), and then performs privileged operations upon that entity if permission is granted. The checking and the access operations should be atomically executed.

For the generation of XML files, a customized patch [5] is applied to GCC 4.2.0. We execute GCC with the option `-fdump-tree-gimple-xml` to create a GIMPLE dump in XML format. Then, our tool parses the XML files in order to generate Remopla model of the source code and combine it with the Remopla representation of the considered security properties. We run our tool on real-world C projects, Figure 7 and Figure 8 give the results of our experiments.

Table 1. Remopla representation of program actions.

<table>
<thead>
<tr>
<th>Program Action</th>
<th>Remopla Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f();</code></td>
<td><code>f();</code></td>
</tr>
<tr>
<td><code>f(v_0,...,v_n);</code></td>
<td><code>ARG[0]=ARG.v_0;...;ARG[n]=ARG.v_n; f();</code></td>
</tr>
<tr>
<td><code>var = v;</code></td>
<td><code>ACTION_VAR_MODIFICATION_var;</code></td>
</tr>
<tr>
<td><code>entry of f()</code></td>
<td><code>ACTION_FUNCTION_CALL_f;</code></td>
</tr>
<tr>
<td><code>return of f()</code></td>
<td><code>ACTION_FUNCTION_RETURN_f;</code></td>
</tr>
<tr>
<td><code>exit();</code></td>
<td><code>ACTION_PROGRAM_END;</code></td>
</tr>
</tbody>
</table>

Table 1. Remopla representation of program actions.
the number of programs that are generated and checked. The third and the fourth column specify the time for constructing the Remopla model and the time for the model-checking, respectively. Notice that the time for constructing and checking program models is property-dependent. We verified the TOCTOU property for all the selected packages. Moreover, we choose to check the file system modules of Linux kernel against the security violation of Figure 2 that has also been reported in SecurityFocus (bid 27796) [16]. That is to demonstrate that our tool can catch real errors.

The fact that the verification takes a long time for some software packages such as OpenSSH is caused by the intensive usage of operations specified in the security automaton by such packages. We are aware of this implementation issue and investigating techniques to solve it. The verification results are shown in Figure 8.

We found two potential TOCTOU vulnerabilities in the OpenSSH package. The following code snippet illustrates one of the detected errors:

```c
// sshpty.c
void pty_setowner(struct passwd *pw, const char *tty)
{
    ...
    if (stat(tty, &st))
        fatal("stat(%.100s) failed: %.100s",
             tty,
             strerror(errno));
    ...
    if (st.st_uid != pw->pw_uid || st.st_gid != gid)
        if (chown(tty, pw->pw_uid, gid) < 0)
            ...
}
```

In this example, a time window exists between the system calls `stat()` and `chown()`. We consider it as a potential vulnerability since the binding of `tty` to a file may change during the race window. We also found one memory access vulnerability in the Linux kernel, that we show in the following source code segment:

```c
// fs/splice.c
static int copy_from_user_mmap_sem(void *dst, const void __user *src, size_t n)
{
    int partial;
    pagefault_disable();
    partial = __copy_from_user_inatomic
               (dst, src, n);
    pagefault_enable();
    ...
}
```

The system call `_copy_from_user_inatomic()` is using the user-space pointer `src` without previous validation with `access_ok()` or `verify_area()` functions.

Notice that our tool produces many false positives since we do not incorporate data flow analysis at this time. Moreover, our analysis is path-insensitive in a sense that we cannot prune infeasible paths to reduce the number of false positives. Like all static verification tools, we face a trade-off between scalability and precision, and we emphasize scalability over the latter.

To ease the analysis of the detected errors, Moped generates a witness path that is represented using an internal scheme. Our tool parses the generated information and maps it to the original source code, producing traces of the detected errors. Figure 9 shows a TOCTOU error trace detected in the OpenSSH package. The source code is represented in HTML format, and the security-relevant operations in the source code are highlighted for easy review.

Figure 6. TOCTOU property
<table>
<thead>
<tr>
<th>Package</th>
<th>LOC</th>
<th>Prog-programs</th>
<th>Security Properties</th>
<th>Remopla Generation (Sec)</th>
<th>Model Checking (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSH 5.0p1</td>
<td>58K</td>
<td>11</td>
<td>TOCTOU</td>
<td>44.9</td>
<td>3136.3</td>
</tr>
<tr>
<td>Postfix 2.5.1</td>
<td>86K</td>
<td>38</td>
<td>TOCTOU</td>
<td>212.5</td>
<td>6184.5</td>
</tr>
<tr>
<td>Apache 1.3.41</td>
<td>75K</td>
<td>9</td>
<td>TOCTOU</td>
<td>14</td>
<td>35.3</td>
</tr>
<tr>
<td>Httpd 2.2.8</td>
<td>210K</td>
<td>12</td>
<td>TOCTOU</td>
<td>23.9</td>
<td>18.2</td>
</tr>
<tr>
<td>Samba 3.0.28a</td>
<td>380K</td>
<td>22</td>
<td>TOCTOU</td>
<td>859.9</td>
<td>19943.6</td>
</tr>
<tr>
<td>Linux Kernel 2.6.23 File-system Modules</td>
<td>486 K</td>
<td>1</td>
<td>TOCTOU</td>
<td>323.5</td>
<td>446.4</td>
</tr>
</tbody>
</table>

**Figure 7. Verification of security properties against real-world C projects**

<table>
<thead>
<tr>
<th>Package</th>
<th>Security Properties</th>
<th>Potential Vulnerabilities</th>
<th>False Positives</th>
<th>Total Findings</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSH 5.0p1</td>
<td>TOCTOU</td>
<td>2</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>Postfix 2.5.1</td>
<td>TOCTOU</td>
<td>—</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Apache 1.3.41</td>
<td>TOCTOU</td>
<td>—</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Httpd 2.2.8</td>
<td>TOCTOU</td>
<td>—</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Samba 3.0.28a</td>
<td>TOCTOU</td>
<td>—</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Linux Kernel 2.6.23 File-system Modules</td>
<td>TOCTOU</td>
<td>—</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 8. Verification results**

**Figure 9. Error trace**
5 Related Work

This section presents approaches and tools based on static analysis and model-checking for vulnerabilities detection in source code.

MOPS is a pushdown model-checking tool for C programs [4]. It provides an automata-based language for the definition of security properties. It only considers control flow information and does not support data flow analysis. MOPS is based on the compiler GCC version 2.x, and can not model check programs that require a newer version of GCC, nor has multi language support.

MetaCompilation (MC) is a static analysis tool that uses a flow-based analysis approach for detecting temporal security errors in C code [9]. With the MC approach, the programmers define their temporal security properties as automata written in a language called metal [3]. The security automata are executed during the traversal of the control flow graph of the analyzed program. A security violation is reported when a security automaton reaches an error state. The expressiveness of the metal language is an appealing feature of the MC approach. In our approach, and we benefit from its high-level of expressiveness. Moreover, our security properties are translated to pushdown systems that have more features than finite state automata, such as counting.

BLAST [10] and SLAM [1] are data-flow sensitive model-checkers based on predicate abstraction. They use an iterative refinement process to locate security violations in source code. Both are mainly used to verify small software of device drivers. Despite the precision of their approach, their iterative process introduces the risk of non-termination.

GMC² [8] is a model-checker for the GCC compiler. As we do, GMC² takes advantage of the TREE-SSA framework and its GIMPLE intermediate representation in order to verify programs implemented in all languages supported by the GCC compiler. The security properties checked are LTL properties defined as procedures or functions. Our model-checking approach is based on the reachability analysis of Moped version 2. The GUI for security property specification, makes our tool more user-friendly than GMC².

CQual [7] is a type-based approach that extends the standard C type system with type qualifiers that are used to express security properties. CQUAL is mainly used to detect information flow properties and requires programmers to manually annotate programs.

As a general statement, we want to emphasize two main features that distinguishes our approach from existing ones. The first one is the natural multi-language support of our approach. The second one is the usage of a conventional model-checker to decouple the verification process from the proper construction of the model to verify. This last feature procures a modular dimension to our approach.

6 Conclusion

In this paper, we present our security verification environment for software systems that brings into a synergy static analysis and model-checking. The synergy consists in utilizing static analysis to automatically build a model-checkable abstraction of the program. It also takes advantage of the model-checking flexibility in verifying a wide range of system-specific security properties. The latter are graphically specified by programmers as finite state automata. Our implementation is based on the TREE-SSA framework of the GCC compiler and the Moped model-checker for pushdown systems. Our conducted experiments showed that our approach can be applied to large software projects. We used our tool to model-check a set of real world programs written in the C language. We were also able to catch real errors in the analyzed packages. Our approach also has the potential to be extended in many directions. The simplicity and the expressiveness of the GIMPLE representation, enables us to extend our approach to all languages supported by the GCC compiler. For now, our approach considers control flow information for security verification. Nevertheless, we plan to utilize the data flow analysis performed by Moped [15] in order to improve our analysis. With Dataflow analysis, we can enlarge the range of security properties to verify. Moreover, it allows us to enhance the precision of our analysis by reducing the number of false positives.

References


