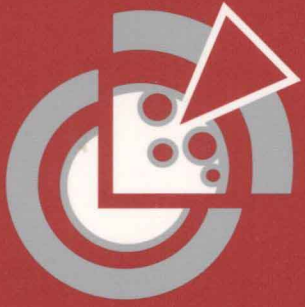
Roland Büschkes
Pavel Laskov (Eds.)

LNCS 4064

Detection of Intrusions and Malware & Vulnerability Assessment

Third International Conference, DIMVA 2006
Berlin, Germany, July 2006
Proceedings



DIMVA 2006

Springer

Roland Büschkes Pavel Laskov (Eds.)

Detection of Intrusions and Malware & Vulnerability Assessment

Third International Conference, DIMVA 2006
Berlin, Germany, July 13-14, 2006
Proceedings

Volume Editors

Roland Büschkes
RWE AG
Opernplatz 1, 45128 Essen
Germany
E-mail: roland.bueschkes@rwe.com

Pavel Laskov
Fraunhofer FIRST
Kekuléstr. 7, 12489 Berlin, Germany
E-mail: pavel.laskov@first.fraunhofer.de

Library of Congress Control Number: 2006928329

CR Subject Classification (1998): E.3, K.6.5, K.4, C.2, D.4.6

LNCS Sublibrary: SL 4 – Security and Cryptology

ISSN 0302-9743
ISBN-10 3-540-36014-X Springer Berlin Heidelberg New York
ISBN-13 978-3-540-36014-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11790754 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 4064

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Lecture Notes in Computer Science

For information about Vols. 1–3979

please contact your bookseller or Springer

Vol. 4072: M. Harders, G. Székely (Eds.), Biomedical Simulation. XI, 216 pages. 2006.

Vol. 4069: F.J. Perales, R.B. Fisher (Eds.), Articulated Motion and Deformable Objects. XV, 526 pages. 2006.

Vol. 4068: H. Schärfe, P. Hitzler, P. Øhrstrøm (Eds.), Conceptual Structures: Inspiration and Application. XI, 455 pages. 2006. (Sublibrary LNAI).

Vol. 4067: D. Thomas (Ed.), ECOOP 2006 – Object-Oriented Programming. XIV, 527 pages. 2006.

Vol. 4066: A. Rensink, J. Warmer (Eds.), Model Driven Architecture – Foundations and Applications. XII, 392 pages. 2006.

Vol. 4064: R. Büschkes, P. Laskov (Eds.), Detection of Intrusions and Malware & Vulnerability Assessment. X, 195 pages. 2006.

Vol. 4063: I. Gorton, G.T. Heineman, I. Crnkovic, H.W. Schmidt, J.A. Stafford, C.A. Szyperski, K. Wallnau (Eds.), Component-Based Software Engineering. XI, 394 pages. 2006.

Vol. 4060: K. Futatsugi, J.-P. Jouannaud, J. Meseguer (Eds.), Algebra, Meaning and Computation. XXXVIII, 643 pages. 2006.

Vol. 4059: L. Arge, R. Freivalds (Eds.), Algorithm Theory – SWAT 2006. XII, 436 pages. 2006.

Vol. 4058: L.M. Batten, R. Safavi-Naini (Eds.), Information Security and Privacy. XII, 446 pages. 2006.

Vol. 4057: J.P. W. Pluim, B. Likar, F.A. Gerritsen (Eds.), Biomedical Image Registration. XII, 324 pages. 2006.

Vol. 4056: P. Flocchini, L. Gąsieniec (Eds.), Structural Information and Communication Complexity. X, 357 pages. 2006.

Vol. 4055: J. Lee, J. Shim, S.-g. Lee, C. Bussler, S. Shim (Eds.), Data Engineering Issues in E-Commerce and Services. IX, 290 pages. 2006.

Vol. 4054: A. Horváth, M. Telek (Eds.), Formal Methods and Stochastic Models for Performance Evaluation. VIII, 239 pages. 2006.

Vol. 4053: M. Ikeda, K.D. Ashley, T.-W. Chan (Eds.), Intelligent Tutoring Systems. XXVI, 821 pages. 2006.

Vol. 4052: M. Bugliesi, B. Preneel, V. Sassone, I. Wegener (Eds.), Automata, Languages and Programming, Part II. XXIV, 603 pages. 2006.

Vol. 4051: M. Bugliesi, B. Preneel, V. Sassone, I. Wegener (Eds.), Automata, Languages and Programming, Part I. XXIII, 729 pages. 2006.

Vol. 4048: L. Goble, J.-J.C.. Meyer (Eds.), Deontic Logic and Artificial Normative Systems. X, 273 pages. 2006. (Sublibrary LNAI).

Vol. 4046: S.M. Astley, M. Brady, C. Rose, R. Zwiggelaar (Eds.), Digital Mammography. XVI, 654 pages. 2006.

Vol. 4045: D. Barker-Plummer, R. Cox, N. Swoboda (Eds.), Diagrammatic Representation and Inference. XII, 301 pages. 2006. (Sublibrary LNAI).

Vol. 4044: P. Abrahamsson, M. Marchesi, G. Succi (Eds.), Extreme Programming and Agile Processes in Software Engineering. XII, 230 pages. 2006.

Vol. 4043: A.S. Atzeni, A. Lioy (Eds.), Public Key Infrastructure. XI, 261 pages. 2006.

Vol. 4042: D. Bell, J. Hong (Eds.), Flexible and Efficient Information Handling. XVI, 296 pages. 2006.

Vol. 4041: S.-W. Cheng, C.K. Poon (Eds.), Algorithmic Aspects in Information and Management. XI, 395 pages. 2006.

Vol. 4040: R. Reulke, U. Eckardt, B. Flach, U. Knauer, K. Polthier (Eds.), Combinatorial Image Analysis. XII, 482 pages. 2006.

Vol. 4039: M. Morisio (Ed.), Reuse of Off-the-Shelf Components. XIII, 444 pages. 2006.

Vol. 4038: P. Ciancarini, H. Wiklicky (Eds.), Coordination Models and Languages. VIII, 299 pages. 2006.

Vol. 4037: R. Gorrieri, H. Wehrheim (Eds.), Formal Methods for Open Object-Based Distributed Systems. XVII, 474 pages. 2006.

Vol. 4036: O. H. Ibarra, Z. Dang (Eds.), Developments in Language Theory. XII, 456 pages. 2006.

Vol. 4035: T. Nishita, Q. Peng, H.-P. Seidel (Eds.), Advances in Computer Graphics. XX, 771 pages. 2006.

Vol. 4034: J. Münch, M. Vierimaa (Eds.), Product-Focused Software Process Improvement. XVII, 474 pages. 2006.

Vol. 4033: B. Stiller, P. Reichl, B. Tuffin (Eds.), Performability Has its Price. X, 103 pages. 2006.

Vol. 4032: O. Etzion, T. Kuflik, A. Motro (Eds.), Next Generation Information Technologies and Systems. XIII, 365 pages. 2006.

Vol. 4031: M. Ali, R. Dapoigny (Eds.), Innovations in Applied Artificial Intelligence. XXIII, 1353 pages. 2006. (Sublibrary LNAI).

Vol. 4029: L. Rutkowski, R. Tadeusiewicz, L.A. Zadeh, J. Zurada (Eds.), Artificial Intelligence and Soft Computing – ICAISC 2006. XXI, 1235 pages. 2006. (Sublibrary LNAI).

Vol. 4027: H.L. Larsen, G. Pasi, D. Ortiz-Arroyo, T. Andreasen, H. Christiansen (Eds.), Flexible Query Answering Systems. XVIII, 714 pages. 2006. (Sublibrary LNAI).

Vol. 4026: P.B. Gibbons, T. Abdelzaher, J. Aspnes, R. Rao (Eds.), Distributed Computing in Sensor Systems. XIV, 566 pages. 2006.

Vol. 4025: F. Eliassen, A. Montresor (Eds.), Distributed Applications and Interoperable Systems. XI, 355 pages. 2006.

Vol. 4024: S. Donatelli, P. S. Thiagarajan (Eds.), Petri Nets and Other Models of Concurrency - ICATPN 2006. XI, 441 pages. 2006.

Vol. 4021: E. André, L. Dybkjær, W. Minker, H. Neumann, M. Weber (Eds.), Perception and Interactive Technologies. XI, 217 pages. 2006. (Sublibrary LNAI).

Vol. 4020: A. Bredenfeld, A. Jacoff, I. Noda, Y. Takahashi (Eds.), RoboCup 2005: Robot Soccer World Cup IX. XVII, 727 pages. 2006. (Sublibrary LNAI).

Vol. 4019: M. Johnson, V. Vene (Eds.), Algebraic Methodology and Software Technology. XI, 389 pages. 2006.

Vol. 4018: V. Wade, H. Ashman, B. Smyth (Eds.), Adaptive Hypermedia and Adaptive Web-Based Systems. XVI, 474 pages. 2006.

Vol. 4016: J.X. Yu, M. Kitsuregawa, H.V. Leong (Eds.), Advances in Web-Age Information Management. XVII, 606 pages. 2006.

Vol. 4014: T. Uustalu (Ed.), Mathematics of Program Construction. X, 455 pages. 2006.

Vol. 4013: L. Lamontagne, M. Marchand (Eds.), Advances in Artificial Intelligence. XIII, 564 pages. 2006. (Sublibrary LNAI).

Vol. 4012: T. Washio, A. Sakurai, K. Nakajima, H. Takeda, S. Tojo, M. Yokoo (Eds.), New Frontiers in Artificial Intelligence. XIII, 484 pages. 2006. (Sublibrary LNAI).

Vol. 4011: Y. Sure, J. Domingue (Eds.), The Semantic Web: Research and Applications. XIX, 726 pages. 2006.

Vol. 4010: S. Dunne, B. Stoddart (Eds.), Unifying Theories of Programming. VIII, 257 pages. 2006.

Vol. 4009: M. Lewenstein, G. Valiente (Eds.), Combinatorial Pattern Matching. XII, 414 pages. 2006.

Vol. 4008: J.C. Augusto, C.D. Nugent (Eds.), Designing Smart Homes. XI, 183 pages. 2006. (Sublibrary LNAI).

Vol. 4007: C. Àlvarez, M. Serna (Eds.), Experimental Algorithms. XI, 329 pages. 2006.

Vol. 4006: L.M. Pinho, M. González Harbour (Eds.), Reliable Software Technologies – Ada-Europe 2006. XII, 241 pages. 2006.

Vol. 4005: G. Lugosi, H.U. Simon (Eds.), Learning Theory. XI, 656 pages. 2006. (Sublibrary LNAI).

Vol. 4004: S. Vaudenay (Ed.), Advances in Cryptology - EUROCRYPT 2006. XIV, 613 pages. 2006.

Vol. 4003: Y. Koucheryavy, J. Harju, V.B. Iversen (Eds.), Next Generation Teletraffic and Wired/Wireless Advanced Networking. XVI, 582 pages. 2006.

Vol. 4001: E. Dubois, K. Pohl (Eds.), Advanced Information Systems Engineering. XVI, 560 pages. 2006.

Vol. 3999: C. Kop, G. Fliedl, H.C. Mayr, E. Métais (Eds.), Natural Language Processing and Information Systems. XIII, 227 pages. 2006.

Vol. 3998: T. Calamoneri, I. Finocchi, G.F. Italiano (Eds.), Algorithms and Complexity. XII, 394 pages. 2006.

Vol. 3997: W. Grieskamp, C. Weise (Eds.), Formal Approaches to Software Testing. XII, 219 pages. 2006.

Vol. 3996: A. Keller, J.-P. Martin-Flatin (Eds.), Self-Managed Networks, Systems, and Services. X, 185 pages. 2006.

Vol. 3995: G. Müller (Ed.), Emerging Trends in Information and Communication Security. XX, 524 pages. 2006.

Vol. 3994: V.N. Alexandrov, G.D. van Albada, P.M.A. Sloot, J. Dongarra, Computational Science – ICCS 2006, Part IV. XXXV, 1096 pages. 2006.

Vol. 3993: V.N. Alexandrov, G.D. van Albada, P.M.A. Sloot, J. Dongarra, Computational Science – ICCS 2006, Part III. XXXVI, 1136 pages. 2006.

Vol. 3992: V.N. Alexandrov, G.D. van Albada, P.M.A. Sloot, J. Dongarra, Computational Science – ICCS 2006, Part II. XXXV, 1122 pages. 2006.

Vol. 3991: V.N. Alexandrov, G.D. van Albada, P.M.A. Sloot, J. Dongarra, Computational Science – ICCS 2006, Part I. LXXXI, 1096 pages. 2006.

Vol. 3990: J. C. Beck, B.M. Smith (Eds.), Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. X, 301 pages. 2006.

Vol. 3989: J. Zhou, M. Yung, F. Bao, Applied Cryptography and Network Security. XIV, 488 pages. 2006.

Vol. 3988: A. Beckmann, U. Berger, B. Löwe, J.V. Tucker (Eds.), Logical Approaches to Computational Barriers. XV, 608 pages. 2006.

Vol. 3987: M. Hazas, J. Krumm, T. Strang (Eds.), Location- and Context-Awareness. X, 289 pages. 2006.

Vol. 3986: K. Stølen, W.H. Winsborough, F. Martinelli, F. Massacci (Eds.), Trust Management. XIV, 474 pages. 2006.

Vol. 3984: M. Gavrilova, O. Gervasi, V. Kumar, C.J. K. Tan, D. Taniar, A. Laganà, Y. Mun, H. Choo (Eds.), Computational Science and Its Applications - ICCSA 2006, Part V. XXV, 1045 pages. 2006.

Vol. 3983: M. Gavrilova, O. Gervasi, V. Kumar, C.J. K. Tan, D. Taniar, A. Laganà, Y. Mun, H. Choo (Eds.), Computational Science and Its Applications - ICCSA 2006, Part IV. XXVI, 1191 pages. 2006.

Vol. 3982: M. Gavrilova, O. Gervasi, V. Kumar, C.J. K. Tan, D. Taniar, A. Laganà, Y. Mun, H. Choo (Eds.), Computational Science and Its Applications - ICCSA 2006, Part III. XXV, 1243 pages. 2006.

Vol. 3981: M. Gavrilova, O. Gervasi, V. Kumar, C.J. K. Tan, D. Taniar, A. Laganà, Y. Mun, H. Choo (Eds.), Computational Science and Its Applications - ICCSA 2006, Part II. XXVI, 1255 pages. 2006.

Vol. 3980: M. Gavrilova, O. Gervasi, V. Kumar, C.J. K. Tan, D. Taniar, A. Laganà, Y. Mun, H. Choo (Eds.), Computational Science and Its Applications - ICCSA 2006, Part I. LXXV, 1199 pages. 2006.

Preface

On behalf of the Program Committee, it is our pleasure to present to you the proceedings of the Third GI SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA). DIMVA is organized by the Special Interest Group Security - Intrusion Detection and Response (SIDAR) of the German Informatics Society (GI) as an annual conference that brings together experts from throughout and outside Europe to discuss the state of the art in the areas of intrusion detection, malware detection and vulnerability assessment.

The DIMVA 2006 Program Committee received 41 submissions from 21 countries. All submissions were carefully reviewed by Program Committee members or external experts according to the criteria of scientific novelty, importance to the field and technical quality. The final selection took place at a Program Committee meeting held on March 10, 2006, in Berlin, Germany. Eleven full papers were selected for presentation and publication in the conference proceedings. In addition, two papers were selected for presentation in the best-practices track of the conference.

The conference took place on July 13-14, 2006, at the conference center of the Berlin-Brandenburg Academy of Sciences in Berlin, Germany. The program featured both theoretical and practical research results, which were grouped into six sessions. Invited talks were given by two internationally renowned security experts: John McHugh, Dalhousie University, Canada, and Michael Behringer, Cisco Systems, France. The conference program was complemented by the European Capture-the-Flag contest CIPHER (Challenges in Informatics: Programming, Hosting and Exploring), a rump session as well as the graduate workshop SPRING, which gave PhD students and young researchers an opportunity to present and discuss their current work and recent results.

We sincerely thank all those who submitted papers as well as the Program Committee members and the external reviewers for their valuable contributions.

For further details please refer to the DIMVA 2006 website at http://www.dimva.org/dimva2006.

July 2006

Roland Büschkes
Pavel Laskov

Organization

DIMVA 2006 was organized by the Special Interest Group Security - Intrusion Detection and Response (SIDAR) of the German Informatics Society (GI), in cooperation with the IEEE Task Force on Information Assurance.

Organizing Committee

General Chair	Pavel Laskov (Fraunhofer FIRST, Germany)
Program Chair	Roland Büschkes (RWE AG, Germany)
Sponsor Chair	Marc Heuse (n.runs, Germany)

Program Committee

Phil Attfield	Northwest Security Institute, USA
Thomas Biege	SUSE LINUX Products GmbH, Germany
Marc Dacier	Institut Eurécom, France
Hervé Debar	France Telecom R&D, France
Sven Dietrich	Carnegie Mellon University, USA
Toralv Dirro	McAfee, Germany
Ulrich Flegel	University of Dortmund, Germany
Dirk Häger	BSI, Germany
Bernhard Hämmerli	HTA Luzern, Switzerland
Oliver Heinz	arago AG, Germany
Peter Herrmann	NTNU Trondheim, Norway
Marc Heuse	n.runs, Germany
Erland Jonsson	Chalmers University of Technology, Sweden
Klaus Julisch	IBM Research, USA
Engin Kirda	Technical University Vienna, Austria
Hartmut König	BTU Cottbus, Germany
Klaus-Peter Kossakowski	DFN-Cert, Germany
Christopher Kruegel	Technical University Vienna, Austria
Jens Meggers	Symantec, USA
Michael Meier	University of Dortmund, Germany
Achim Müller	Deutsche Telekom Laboratories, Germany
Martin Naedele	ABB Corporate Research, Switzerland
Dirk Schadt	Computer Associates, Germany
Robin Sommer	ICIR/ICSI, USA
Axel Tanner	IBM Research, Switzerland
Marco Thorbrügge	ENISA, Greece
Stephen Wolthusen	Gjøvik University College, Norway

External Reviewers

Magnus Almgren	Chalmers University of Technology, Sweden
Nenad Jovanovic	Technical University Vienna, Austria
Corrado Leita	Institut Eurécom, France
Andreas Moser	Technical University Vienna, Austria
Sebastian Schmerl	BTU Cottbus, Germany
Olivier Thonnard	Institut Eurécom, France

Steering Committee

Chairs	Ulrich Flegel (University of Dortmund, Germany)
	Michael Meier (University of Dortmund, Germany)
Members	Roland Büschkes (RWE AG, Germany)
	Marc Heuse (n.runs, Germany)
	Klaus Julisch (IBM Research, USA)
	Christopher Kruegel (Technical University Vienna, Austria)

Sponsoring Institutions

Table of Contents

Code Analysis

Intrusion Detection

Threat Protection and Response

Malware and Forensics

Deployment Scenarios

Using Type Qualifiers to Analyze Untrusted Integers and Detecting Security Flaws in C Programs

Ebrima N. Ceesay, Jingmin Zhou, Michael Gertz, Karl Levitt, and Matt Bishop

Computer Security Laboratory
University of California at Davis
Davis, CA 95616, USA
{ceesay, zhouji, gertz, levitt, bishop}@cs.ucdavis.edu

Abstract. Incomplete or improper input validation is one of the major sources of security bugs in programs. While traditional approaches often focus on detecting string related buffer overflow vulnerabilities, we present an approach to automatically detect potential integer misuse, such as integer overflows in C programs. Our tool is based on CQual, a static analysis tool using type theory. Our techniques have been implemented and tested on several widely used open source applications. Using the tool, we found known and unknown integer related vulnerabilities in these applications.

1 Introduction

Most known security vulnerabilities are caused by incomplete or improper input validation instead of program logic errors. The ICAT vulnerability statistics [1] show for the past three years that more than 50% of known vulnerabilities in the CVE database are caused by input validation errors. This percentage is still increasing. Thus, improved means to detect input validation errors in programs is crucial for improving software security.

Traditionally, manual code inspection and runtime verification are the major approaches to check program input. However, these approaches can be very expensive and have proven ineffective. Recently, there has been increasing interest in static program analysis techniques and using them to improve software security. In this paper, we introduce a type qualifier based approach to perform analysis of user input integers and to detect potential integer misuse in C programs. Our tool is based on CQual [2], an extensible type qualifier framework for the C programming language.

An integer is mathematically defined as a real whole number that may be positive, negative, or equal to zero [3]. We need to qualify this definition to include the fact that integers are often represented by integer variables in programs. Integer variables are the same as any other variables in that they are just regions of memory set aside to store a specific type of data as interpreted by the programmer [4]. Regardless of the data type intended by the programmer, the computer interprets the data as a sequence of bits. Integer variables on various systems may have different sizes in terms of allocated bits. Without loss of generality, we assume that an integer variable is stored in a 32-bit memory location, where the first bit is used as a sign flag for the integer value.

Integer variables are widely used in programs as counters, pointer offsets and indexes to arrays in order to access memory. If the value of an integer variable comes

R. Büschkes and P. Laskov (Eds.): DIMVA 2006, LNCS 4064, pp. 1–16, 2006.

from untrusted source such as user input, it often results in security vulnerabilities. For example, recently an increasing number of integer related vulnerabilities have been discovered and exploited [5, 6, 7, 8, 9]. They are all caused by the misuse of integers input by a user. The concept of integer misuse like integer overflow has become common knowledge. Several researchers have studied the problem and proposed solutions like compiler extension, manual auditing and safe C++ integer classes [4, 10, 11, 12, 13, 14]. However, to date there is no tool that statically detects and prevents integer misuse vulnerabilities in C programs.

Inspired by the classical Biba Integrity Model [15] and Shankar and Johnson's tools [3, 16] to detect format string and user/kernel pointer bugs, we have implemented a tool to detect potential misuse of user input integers in C programs. The idea is simple: we categorize integer variables into two types: *trusted* and *untrusted*. If an *untrusted* integer variable is used to access memory, an alarm is reported. Our tool is built on top of CQual, an open source static analyzer based on the theory of type qualifiers [2]. Our experiments show that the tool can detect potential misuse of integers in C programs.

The rest of the paper is organized as follows: Section 2 gives a brief introduction to CQual and the theory behind it. Section 3 describes the main idea of our approach and the development of our tool based on CQual. Section 4 shows the experiments we have performed and the results. In Section 5 we discuss several issues related to our approach. Section 6 discusses related work. Finally, Section 7 concludes this paper with future work.

2 CQual and Type Qualifiers

We developed our tool as an enhancement to CQual. It provides a type-based static analysis tool for specifying and checking properties of C programs.

The idea of type qualifiers is well-known to C programmers. Type qualifiers add additional constraints besides standard types to the variables in the program. For example, in ANSI C, there is a type qualifier *const* that attaches the unalteration property to C variables. However, qualifiers like *const* are built-in language features of C, which seriously restrict the scope of their potential applications. CQual allows a user to introduce new type qualifiers. These new type qualifiers specify the customized properties in which the user is interested. The user then annotates a program with new type qualifiers, and lets CQual statically check it and decide whether such properties hold throughout the program. The new type qualifiers introduced in the program are not a part of the C language, and C compilers can ignore them.

There are two key ideas in CQual: *subtyping* and *type inference*.

Subtyping is familiar to programmers who practice object-oriented programming. For example, in GUI programming, a class DialogWindow is a subclass of class Window. Then we say DialogWindow is a subtype of Window (written as DialogWindow $\leq$ Window). This means that an object of DialogWindow can appear wherever an object of Window is expected, but not vise versa. Thus, if an object of type Window is provided to a program where a DialogWindow is expected, it is a potential vulnerability and the program does not type check.

CQual requires the user to define the subtyping relation of user supplied type qualifiers. The definition appears as a lattice in CQual's `lattice` configuration file. For

example, if we define the lattice for type qualifiers Q_1 and Q_2 as: $Q_1 \leq Q_2$, it means for any type τ, $Q_1\tau$ and $Q_2\tau$ are two new *qualified types*, and $Q_1\tau$ is a subtype of $Q_2\tau$ (written as $Q_1\tau \leq Q_2\tau$) [2, 3]. Thus, a variable of type $Q_1\tau$ can be used as a variable of type $Q_2\tau$, but not vise versa.

Manually annotating programs with type qualifiers can be expensive and error prone. Therefore, CQual only requires the user to annotate the programs at several key points and uses *type inference* to automatically infer the types of other expressions. For example, in the following code fragment, the variable `b` is not annotated with the qualifier *untrusted*, but we can infer this qualifier for `b` from the assignment statement [1].

```
int $untrusted    a;
int               b;
...
b = a;
```

To eliminate the burden of annotating programs across multiple source code files, CQual provides a `prelude` file. A user can define fully annotated function declarations in this file, and let CQual load it at run-time. This is particularly useful when the source code of certain functions is not available, e.g., the library functions and system calls. In this situation, CQual is still able to use type inference to infer the *qualified types* of expressions from the annotations in the `prelude` file. For example, in the following code fragment, after we annotate the C library function `scanf` in the `prelude` file, CQual is able to infer that the variable `a` is an *untrusted* integer variable in the program.

```
prelude:
   int scanf (char* fmt, $untrusted ...);

user_program.c:
   int    a;
   scanf ("%d", &a);
```

3 Integer Misuse Detection

This section describes how our tool detects potential integer misuse vulnerabilities in C programs. Inspired by the Biba Integrity Model [15], we propose a security check tool based on CQual to detect integer misuse. In our tool, security holes are detected by tracing dependency of variables. Integrity denotes security level. If a value of a variable is updated by an *untrusted* variable during the execution of a program, then the integrity of the variable decreases and the value is regarded as *untrusted*.

Therefore, we categorize integer variables in programs into two types: *trusted* and *untrusted*. An integer variable is *untrusted* because either its value is directly fetched from user input, or the value is propagated from user input. An integer variable is *trusted* because its value has no interaction with *untrusted* integers. In addition, we define program points that generate and propagate *untrusted* integer variables, and program points

[1] CQual requires the type qualifiers start by a $ sign. For convenience, we ignore the $ sign in our discussion except for the code fragments.

that should only accept *trusted* integer variables. For example, suppose each integer parameter of a function `read_file` is annotated as *trusted*. If there is a flow in a program that an *untrusted* integer variable is used as a parameter of function `read_file`, a security exception is generated, resulting in a alarm.

In order to speed up our efforts and develop a working prototype several assumptions are made.

3.1 Assumptions

First, we assume that a programmer does not deliberately write erroneous code. This means that we trust the integer variables prepacked in programs if these internal integer variables do not have any direct or indirect relations with user input. For example, an integer variable may be initialized statically in a program and it is used as index to access an array. There is no interaction between this integer variable and user input. The assumption is that the programmer knows the exact size of the array being accessed and the value of this integer variable is not larger than boundary of the array. We believe that this is a reasonable assumption. In fact, this kind of assumptions are often needed for many static analysis techniques.

We also assume that integer misuse only happens when *untrusted* integer variables are used to access memory. This means it is safe to use *untrusted* integer variables in many other situations. This is because, to the best of our knowledge, most integer related vulnerabilities are only associated to memory access.

To make it clear, user input integers are not limited to the integers given to an application by a command line option, or typed in by a user at a program prompt. They also include many other methods by which a program obtains data from outside the program itself, such as reading a file or receiving network packets. User input data in the context of this paper means the data that is not prepackaged within the program.

3.2 New Type Qualifiers

The first step is to define the type qualifiers for integer variables and the lattice of these type qualifiers in CQual's `lattice` file. Since there are two categories of integer variables in our method, two type qualifiers are defined: *untrusted* and *trusted*. These two qualifiers have a sub-typing relation of *trusted* $\leq$ *untrusted*. This implies that programs that accept an *untrusted* integer variable can also accept a *trusted* integer variable. However, the reverse is not true.

Our implementation is not limited to integer variables and we apply the two new qualifiers to any types of variable in C programs. This is particularly important since integers are often converted from other types of data, and we keep track of these changes. As shown in the following code fragment, the integer variable `a` will become *untrusted* after the assignment because the content of string `str` is *untrusted*[2], and the declaration of the function `atoi` in the `prelude` file specifies that an *untrusted* string has been converted to an *untrusted* integer.

[2] Different positions of a qualifier for a pointer variable have different meanings. In particular, `char untrusted *buf` defines the memory content pointed by `buf` as *untrusted*, `char* untrusted buf` defines the pointer variable `buf` itself as *untrusted*.

```
prelude:
    int $untrusted atoi (char $untrusted* string);

user_program.c:
    char $untrusted* str;
    int              a;
    ...
    a = atoi (str);
```

3.3 Annotations with Type Qualifiers

The second step is to determine the source of *untrusted* data in programs and how they propagate in the programs, and annotate the programs using the *untrusted* qualifier.

By our definition, all user inputs are *untrusted*. Therefore, we need to identify all locations that accept data from outside the programs. For programs based on standard C library and UNIX system calls, the sources of *untrusted* data include: program argument array `argv`, environment variables, standard I/O input, files and network sockets. Program argument array `argv` and environment variables accept user supplied parameters; standard I/O input is usually used to accept keyboard input from the user; files store the data from the file systems; and network sockets provide data transmitted over the network. In POSIX compatible systems, most inputs are handled in the same way as files, so it is unnecessary to distinguish them. Thus identification of user input is relatively simple: find all C library functions and system calls that are related to files, and pick those that fetch data. For example, the system call `read` and C library function `fread` both read data from files. We annotate them in the `prelude` file as illustrated in the following code fragment. In these declarations, the pointer `buf` points to a memory buffer that saves the input data. This memory buffer is annotated as *untrusted*.

```
prelude:
 int read (int fd, void untrusted* buf, int);
 int fread (void untrusted *buf, int, int, FILE*);
```

We focus on a specific type of *untrusted* data: integer variables. Thus, it is necessary to determine type conversion from *untrusted* data to *untrusted* integers. The standard C library provides a limited number of functions that can generate integers from strings. We categorize them into two groups:

1. General purpose library functions that can convert strings into integers. These functions include group of `scanf` functions, e.g., `scanf`, `fscanf`, `sscanf`, etc.. They use the "%d" format to convert a string into an integer.
2. Single purpose library functions that convert strings into integers. These functions include `atoi`, `atol`, `strtol`, `atof`, etc.

In group one, since `scanf` and `fscanf` directly read in data from user input, the integer variables fetched are immediately annotated as *untrusted*. However, since the first argument of `sscanf` can either be *trusted* or *untrusted*, the annotation of its fetched

variables will depend on the qualifier of the first argument. This difference is shown in the following code fragment [3]:

```
prelude:
    int scanf (char* fmt, untrusted ...);
    int fscanf(FILE*, char* fmt, untrusted ...);
    int sscanf(char $_1* str, char* fmt, $_1_2 ...);
```

The functions in the second group are similar to `sscanf`: the qualifier of the returned integer variable depends on the qualifier of the input string. This is shown in the code fragment below:

```
prelude:
    int $_1 atoi (char $_1* s);
    long $_1 atol (char $_1* s);
    long $_1 strtol (char $_1* s);
```

In addition to C library string functions, there are two other methods that convert different types of data into integers. One is type cast. For example, a character variable `ch` may be cast into an integer variable and be assigned to an integer variable `a`. In this case, CQual automatically propagates the type qualifiers of `ch` to `a`. In the other case integers are fetched directly into a memory location of an integer variable. For example, a program can call function `fread` to fetch data from a file into a buffer that is the memory address of an integer variable. In this case, since the content of the buffer is annotated as *untrusted*, CQual will infer the integer variable as *untrusted*.

We must consider the propagation of *untrusted* data in addition to the source of these data. CQual uses type inference to automatically infers the propagation of type qualifiers between variables through assignments. However, this is often inadequate in practice. For example, source code of library functions is often unavailable during analysis. If these functions are not annotated, propagation in libraries would be missed. Such library functions include `strcpy`, `strncpy`, `memcpy`, `memmove`, etc.. We must annotate these functions as below:

```
prelude:
    char $_1_2* strcpy(char $_1_2*, char $_1*);
    char $_1_2* strncpy(char $_1_2*, char $_1*, size_t);
    void $_1_2* memcpy(void $_1_2*, void $_1*, size_t);
    void $_1_2* memmove(void $_1_2*, void $_1*, size_t);
```

After identifying the source of *untrusted* integer variables, the next step is to determine that all expressions that must accept *trusted* integers, and make annotation as needed. To enforce memory safety, all integer variables used as direct or indirect offsets of a pointer must be *trusted* integers.

[3] `$_1` and `$_1_2` are polymorphic qualifier variables in CQual. CQual treats each pair of polymorphic variables (`A`, `B`) as if there was an assignment from `A` to `B` when `A` is a substring of `B`.