Securing Legacy C Applications Using Dynamic Data Flow Analysis

Steve Cook
Southwest Research Institute

Dr. Calvin Lin and Walter Chang
University of Texas at Austin

Most attacks on networks and information systems consist of exploits of software vulnerabilities. Even safe programming languages are not immune to problems such as Structured Query Language (SQL) injection, cross-site scripting, or other information flow-related vulnerabilities. While current technological and educational efforts seek to ensure the security of future applications, millions of lines of existing software must be secured as we work to defend our national infrastructure. What is needed is an automatic and scalable method that identifies and traps runtime exploits and that can update existing software as security policies evolve. This article presents ongoing research on an approach to securing legacy C applications, while remaining applicable to future problems and languages.

Most attacks on networks and information systems begin by exploiting a vulnerability in a software application that is resident on a host computer, server, or even an appliance designed to provide network defense. Addressing these vulnerabilities is currently very labor-intensive, requiring constant updates and patches. All of us have become accustomed to receiving software security updates on an almost daily basis for many of our commonly used applications. Terms such as denial of service, phishing, botnets, and spamming are all becoming part of our everyday vernacular, and our collective concern.

Current technological and educational efforts seek to ensure the security of future applications. Most approaches to resolving today's security concerns focus on single-point solutions. Furthermore, millions of lines of existing software that comprise our legacy systems must be secured to defend those information systems on which our national infrastructure depends. Unfortunately, today's software engineers are not typically trained in the development of secure software systems. Implementation of application security requires that the programmer be an expert not only in the application domain, but also in secure coding practices. Our universities are beginning to add security to the software educational curriculum so that new graduates can distinguish good practices from bad as they translate software designs into source code. The Department of Homeland Security has created a publicly available Web site to capture best practices in developing secure software at <https://buildsecurityin.us-cert.gov>. However, even when a programmer is trained in best practices for secure programming, it is unrealistic to depend upon the programmer to develop code that is absent of vulnerabilities.

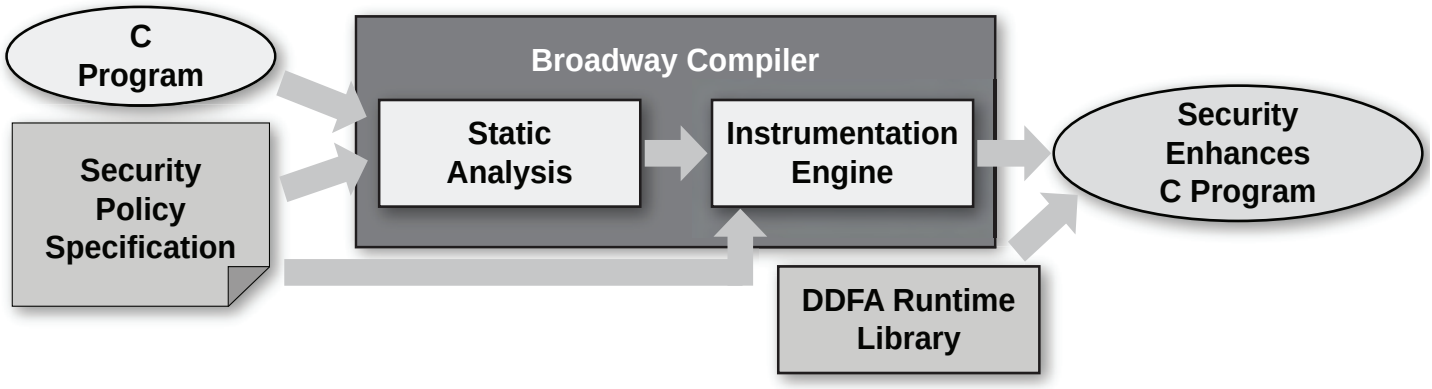
To supplant the ad-hoc security enhancing efforts of today and meet the challenges of tomorrow, an automated method is needed to identify and guard against runtime exploits while remaining flexible and agile as security policies change and evolve. In this article, we present research work in progress to develop a system that ensures that C programs enforce a wide variety of user-defined security policies with a minimum of runtime overhead and disruption to development processes. In the future, our system can be extended to handle multiple languages and complement new security solutions.

What Is Dynamic Data Flow Analysis (DDFA)?

DDFA is an extensible, compiler-based system that automatically instruments the source code of arbitrary (meaning without any assumptions on the code) C programs to enforce a user-specified security policy. The system does not require any modification to the original source code by the developer and also does not significantly degrade a program's runtime performance. Moreover, it has the ability to simultaneously enforce many different classes of security vulnerabilities.

The DDFA system is built upon the Broadway static data flow analysis and error checking system, which is a source-to-source translator for C developed by the computer sciences department at the University of Texas at Austin (UT-Austin) [1]. UT-Austin and the Southwest Research Institute (SwRI) are collaborating to enhance the Broadway specification language and analysis infrastructure with a dependence analysis, an instrumentation

Figure 1: *Overall Architecture of DDFA System*



engine, and a dynamic data flow library.

Figure 1 shows the overall architecture of the DDFA system. Input to the Broadway compiler consists of the source code of an untrusted program and a security policy specification file. The output is an enhanced version of the original source code that has been automatically instrumented with DDFA runtime library calls. The modified program is then compiled for the platform of choice so that its security policy can be enforced at runtime using DDFA. The system does not require hardware or operating system changes.

The primary design goals are the following:

1. Minimize the required developer involvement.
2. Minimize runtime overhead of the secured program.
3. Provide sufficient generality for multi-level security support and enough extensibility for future capabilities.

Minimizing developer involvement is achieved through a security policy specification file that is independent of the program. A security policy is defined once by a security expert using a simple language, which has a direct mapping to the application programming interface to which the program is written. The policy, once defined, can be applied to many different programs. The DDFA approach is easily integrated into the development workflow, adding only an additional compilation step before application deployment.

To minimize the runtime overhead of an executing program, the DDFA approach builds on the body of research in static analysis and leverages semantic information provided by the security policy to enable optimizations beyond standard compiler techniques. This results in a program that is instrumented with additional code only where *provably necessary*, so innocuous flows of data are not tracked at runtime, thus keeping runtime overhead low.

For sufficient generality and extensibility in security exploits, a DDFA approach is used instead of a more traditional dynamic taint analysis [2, 3, 4]. Taint analysis tracks the flow of tainted data (i.e., data originating from the potential attacker) through the system at runtime and then checks that the tainted data is not misused. Our approach expands on the generality of taint tracking by recognizing that taint tracking is a special case of data flow tracking (DDFA). Whereas taint analysis typically tracks one bit of information, data flow analysis can track multiple bits of information and can combine the information in more flexible ways than taint analysis. This allows our approach to support multi-level security and provide a higher level of generality that could be used for other unanticipated security challenges in the future.

Traditional Tainted Data Attacks	Security Problems Taint Tracking Cannot Handle
Format String Attacks	File Disclosure Vulnerabilities
SQL Injection	Labeled Security Enforcement
Command Injection	Role-Based Access Control
Cross-Site Scripting	Mandatory Access Control
Privilege Escalation	Accountable Information Flow

Table 1: *Vulnerabilities That Taint Tracking Can and Cannot Handle*

Table 1 identifies exploits that can and cannot be handled by taint-based systems. Since the DDFA approach supports general data flow tracking, it can handle all of these exploits, and can do so simultaneously. Such generality will become particularly important as developers move to memory-safe languages such as Java and C#, where the use of taint tracking to enforce secure control flow is not as important. Security vulnerabilities that are not currently addressed by the DDFA approach include those such as covert timing channel vulnerabilities, since they are outside the scope of data flow analysis [5].

"To supplant the ad-hoc security-enhancing efforts of today and meet the challenges of tomorrow, an automated method is needed to identify and guard against runtime exploits while remaining flexible and agile as security policies change and evolve."

How Does DDFA Work?

In this section, we will discuss in more detail the components of the DDFA system and how they work. The components include the following: the *policy specification*, *static data flow analysis*, *instrumentation engine*, and *DDFA*.

Policy Specification

The policy specification is defined using the Broadway specification language [1, 6]. It is a simple declarative language that is used to tell the compiler how to perform specific data flow analysis by supplying rules. The two high-level items to specify are the *lattice* definition and the *summaries* for the library functions and system calls.

A lattice must be defined for each type of analysis to be performed. For example, in a format string attack, taint analysis can be used. A lattice called *Taint* would be defined with two flow values which could be named *Tainted* and *Untainted*. Furthermore, *Untainted* is placed higher than *Tainted* to signify that when the two flow values are combined on a particular object being tracked, the result is the lower of the two, *Tainted*. Lattices naturally model hierarchical security levels and are ideal for reasoning about multilevel security and access control that go beyond taint tracking, such as role-based access control (RBAC) [7].

Summaries for the library functions and system calls define how the lattice flow values are introduced into the system, how the flow values propagate through the system, and how to track unsafe use of the lattice flow values. Continuing our format string attack example, the following specification declares that data introduced into the system through the network library call *recv()* would always be tainted:

```
procedure recv(s, buf, len, flags)
{
 on_entry { buf → buffer }
 analyze Taint { buffer ← Tainted }
}
```

Here, the *on_entry* keyword describes function parameters relevant to the analysis

and gives a name, *buffer*, to the object pointed to by *buf*. The *analyze* keyword describes the effect of the function on lattice flow values.

To allow the compiler to reason about the propagation of tainted data, the following specification of the library system call *strdup()* declares that it returns a pointer to *string_copy* and that *string_copy* should have whatever taintedness that *string* has:

```
procedure strdup(s)
{
 on_entry { s → string }
 on_exit { return → string_copy }
 analyze Taint { string_copy ← string }
}
```

Finally, to track the unsafe use of tainted data, an error handler would be specified as follows in the summary for a library call such as *printf()* (an ultimate perpetrator in a format string attack) when unsafe data is actually used:

```
procedure printf(format, args)
{
 on_entry { format → format_string }
 error_if (Taint : format_string could-be
          Tainted)
    fsv_error_handler();
}
```

Here, the *error* keyword signals to the compiler that special action is required when the condition is met.

Unlike most taint tracking systems, the security policy (lattice and summary declarations) and underlying analysis is not hard-coded into the compiler or runtime system. Instead, this semantic information for the analysis is provided in a separate specification file. This separation allows the system to remain general and flexible enough to enforce a wide variety of security policies.

Some insight into the generality and flexibility of the DDFA system can be demonstrated by how it guards against *file disclosure* attacks which cannot be handled with traditional taint tracking systems. For the analysis, two lattices must be defined as follows to track the origin of data and its trustedness:

```
property Kind : { File, Filesystem, Client,
                  Server, Pipe, Command,
                  StandardIO,
                  Environment,
                  SystemInfo, NameServer
                  }

property Trust : {Remote, External,
                  Internal }
```

For a true file disclosure problem, only *File* is used, but this definition could be reused for other policies that need to distinguish between other sources. For the sake of brevity, summaries for how lattice flow values are introduced into the system and how those flow values are propagated will not be shown here, as it is somewhat similar to the previous format string attack example. The following specification defines a violation of the policy where *File* data from a file with *Remote* trustedness is sent to a *server* socket with *Remote* trust (indicating that it was initiated by a remote user):

```
procedure write(fd, buf_ptr, size)
{
 on_entry { fd → IOHandle, buf_ptr →
        buffer }
 error if ( (Trust : buffer could-be Remote
   && Kind : buffer could-be File) &&
   (Trust : IOHandle could-be Remote
   && Kind : IOHandle could-be Server) )
     file_disclosure_error_handler()
}
```

Static Data Flow Analysis

To avoid the cost of tracking all objects at runtime, the DDFA system performs inter-procedural data flow analysis that identifies all program locations where policy violations might occur. A subsequent inter-procedural analysis identifies all program statements that affect the lattice flow value of objects that may trigger a policy violation. Both of these compiler passes are supported by a fast and precise pointer analysis for potentially aliased data.

The first pass statically identifies all possible violations of the security policy. If the analysis can prove that there are no such vulnerabilities in the program, no further analysis or instrumentation is needed. When analysis determines that a vulnerability exists or cannot determine if a vulnerability is genuine, additional analysis is needed to determine where instrumentation is required.

The second pass is an inter-procedural dependence analysis that identifies all the statements in the program that affect the flow value of objects that may trigger a security violation. This pass provides a list of statements to the instrumentation engine to be inserted into the original source code so that dynamic data flow analysis can be performed at runtime.

Pointer analysis is necessary for memory-unsafe languages like C and C++ because objects could have many different pointers pointing to them, making it trickier to reason precisely about the flow of data. The limited scalability of pointer analysis has stymied previous attempts to apply inter-procedural analysis to dynamic taint tracking [8]. However, the DDFA system makes use of a highly scalable and accurate client-driven pointer analysis that leverages the semantic information provided by the security policy to dramatically reduce the amount of code that needs to be instrumented [9, 10].

Instrumentation Engine

The instrumentation engine uses the results from static data flow analysis to determine where in the program additional code must be inserted to support the DDFA at runtime. The instrumentation engine also uses the semantics of the security policy specification to determine which particular calls from the DDFA runtime library will be inserted into the program. Continuing our format string attack example, the following C code snippet shows how the network library call *recv()* would be instrumented if static analysis determines that it may introduce suspect data into the system:

```
recv(sock_fd, recv_msg, 10, 0);
ddfa_insert(DDFA_LTAINT, recv_msg,
          strlen(recv_msg),DDFA_
          LTAINT_TAINTED);
```

The *ddfa_insert()* call sets the memory region occupied by the string object *recv_msg* as *TAINTED*. Any attempt to copy the data occupied by this object to another memory region or to use this data in an unsafe manner would pass the flow value tainted to the copied object or trigger the error handler respectively. To support propagation of tainted data, the following code snippet shows how the library call *strdup()* would be instrumented:

```
copy_msg = strdup(recv_msg);
ddfa_copy_flowval(DDFA_LTAINT,
copy_msg, recv_msg, strlen(copy_msg));
```

Finally, the following code snippet shows how the library call *printf()* would be instrumented to conditionally invoke an error handler before unsafe use of tainted data occurs:

```
if ( ddfa_check_flowval(DDFA_LTAINT,
  copy_msg, DDFA_LTAINT_TAINTED) )
  {fsv_error_handler();}
printf(copy_msg);
```

This example illustrates how the DDFA approach can go much further than simply detecting vulnerabilities, as with purely static approaches, by allowing the original program to execute securely

even if the programmer does not fix the underlying problem. Additionally, the flow of the original program is not affected unless a potential unsafe use of tainted data is detected.

DDFA

The DDFA is supported by the data flow analysis at runtime library and, for each type of analysis, tracks the lattice flow values for all objects identified by the static analysis as potentially unsafe. As the program executes, our system tracks object flow values at the byte level, which provides fine-grained tracking that is necessary in memory unsafe languages such as C. If an unsafe use of an object occurs, the error handler specified in the security policy will be invoked before the object is actually used in an unsafe manner. The primary benefit of performing data flow analysis at runtime is to subvert a security attack by invoking the appropriate error handler specified in the security policy before an unsafe use of an object occurs.

Effectiveness of the DDFA Approach

In this section we will address, in both objective and subjective terms, the effectiveness of the DDFA approach.

Minimizing Impact on Development Processes

One of the most important benefits in using the DDFA system is that it works on existing C programs and does not require the developer to make any changes to the original source code. The developer need only recompile their program with the DDFA system to create a security-enhanced version of their original source code. If not using one of the default policy specifications provided by our system, such as *taint tracking* or *file disclosure vulnerability*, a security expert can extend the system by creating a new policy specification file. In contrast, to extend a conventional taint tracking system to something other than taint tracking, core components of the compiler infrastructure would have to be rewritten. In the DDFA system, the only change is in the policy specification file itself. This also allows new security policies to be developed quickly in response to new attacks, resulting in a more agile response to the ever-changing security environment.

Although the DDFA approach currently requires the source code of the application as input to the system instead of the binary executable, it does not require the implementation source code of the library functions and system calls that appear as summaries in the policy specification file. Only an understanding of the behavior of the function calls is required. Moreover, the work described in this article is part of an ongoing research program, which has as a longer-term goal of applying the DDFA approach to binary executables. This could be an evolutionary step in the research, since the DDFA system already performs its analysis on a low-level representation of the input program. However, there are challenges in moving to binary because significant information that is leveraged in minimizing the performance impacts of security insertion is lost in the compilation process.

Program	Original	DDFA	Runtime Overhead
pfingerd	3.07 seconds (s)	3.19 s	**3.78 %**
muh	11.23 milliseconds (ms)	11.23 ms	**0.0 %**
wu-ftpd	2.745 megabytes per second (MB/s)	2.742 MB/s	**< 1 %**
bind	3.580 ms	3.566 ms	**< 1%**
apache	6.062 MB/s	6.062 MB/s	**< 1%**
		Average Overhead:	**0.65 %**

Table 2: *Runtime Overhead for Server Programs Performing Taint Analysis*

"One of the most important benefits in using the DDFA system is that it works on existing C programs and does not require the developer to make any changes to the original source code."

Another related aspect of our system is that the security enforcement is added to the program after it has been developed. This opens up many new software engineering possibilities such as code separation. In-house software will be more maintainable and agile because it is unfettered by security concerns. Likewise, commercial off-the-shelf and open source software can be brought into a highly trusted environment and then automatically made more secure. It also allows an organization to keep their security policy specification private when subcontracting software development to outside organizations.

Minimizing Performance Overhead

Another important benefit is that the DDFA approach takes advantage of the fact that only a very small portion of a program is actually involved in any given security attack [11]. Identifying this small portion of the program, however, requires sophisticated static analysis. Without this analysis, large portions of the program would have to be instrumented, substantially increasing the program's runtime overhead.

In order to quantify the runtime overhead that is incurred on programs enhanced by the DDFA system, we measured the runtime overhead for two different sets of *open-source* C programs. In the first set, we measured response time or throughput for several different server type programs with considerable input/output. We took measurements against the original program and then again after instrumentation by the DDFA system (in this case, applying a taint-based policy specification). As shown in Table 2, our solution has an average overhead of 0.65 percent. The current fastest compiler-based and dynamically optimized systems report server application overhead of 3-7 percent, and 6 percent, respectively [12, 13].

In the second test set, we measured Standard Performance Evaluation Corporation (SPEC) workloads of four SPECint 2000 benchmarks that are compute-bound applications after performing a format string vulnerability analysis. In each case, our static analysis determines that the programs contain no such vulnerability, as expected. Thus, the true overhead for these examples is zero percent. In order to understand the impact of our system on compute-bound programs that do contain vulnerabilities, we manually inserted a vulnerability into each of the benchmarks. As can be seen in Table 3 (see next page), the average runtime overhead is less

Program	Runtime Overhead
gzip	51.35 %
vpr	< 1 %
mcf	< 1%
crafty	< 1%
Average:	12.9 %

Table 3: *Runtime Overhead for Compute-Bound Programs Performing Format String Vulnerability*

than 13 percent, which is significantly better than the best previously reported averages of 75-260 percent [12, 13].

Minimizing Code Expansion

Because our system adds instrumentation to the source program, it expands the programs static code size. To quantify this property, we measured code expansion by comparing the sizes of the original and modified binary executables after performing taint tracking on the server programs mentioned previously. As can be seen in Table 4, the average expansion is less than 1 percent. Compiler-based systems such as the GNU Image-Finding Tool report 30-60 percent increases in binary size [8].

Related Capabilities and Future Directions

Finally, another important benefit of the DDFA system is that the technology is applicable to future threats and other areas that are not specific to security. For example, systems that depend on semantic information such as information flow (i.e., privacy concerns) or access controls can be enhanced by our system without modifying the core infrastructure. We recently demonstrated this generality by showing that the policy specification file can be used to define roles in a RBAC system, and then subsequently applied to a set of software that previously did not have RBAC. The DDFA system can go beyond problems such as buffer overflows and overwrite attacks that continue to plague legacy languages and solve problems affecting even safer languages such as Java and C#. These languages are not immune to attacks like SQL injection and cross-site scripting that depend on semantic, not language-level, errors in data handling. DDFA, along with the semantic information captured by the policy specification, can address these types of problems.

Our challenge, therefore, is to develop solutions that can both be applied to existing legacy software today for immediate benefits while also looking forward to the more sophisticated challenges that face us in the future. We believe that the DDFA system is well positioned to provide a practical approach to enhancing the security of legacy software in the near future. We are also continuing our research in increasing the scalability of pointer analysis, integrating language-independence into the DDFA technology, as well as researching and testing the breadth of applicability of the approach itself.◆

Table 4: *Static Code Expansion After Performing Taint Tracking*

Program	Original (bytes)	DDFA (bytes)	Code Overhead
pfingerd	49,655	49,655	**0.0 %**
muh	59,880	60,488	**1.01 %**
wu-ftpd	205,487	207,997	**1.22 %**
bind	215,669	219,765	**1.90 %**
apache	552,114	554,514	**0.43 %**
		Average Overhead:	**0.91 %**

Acknowledgements

This work is funded in part by the Intelligence Advanced Research Projects Activity National Intelligence Community Enterprise Cyber Assurance Program. Additionally, the authors would like to acknowledge the following staff who form part of this research team: Ben Hardekopf and Brandon Streiff (of UT-Austin); Mark Brooks, Nakul Jeirath, Arif Kasim, Ronnie Killough, Jeremy Price, and Galen Rasche (all of the SwRI).

References

1. Guyer, S.Z. "Incorporating Domain-Specific Information into the Compilation Process." Diss., The University of Texas at Austin, Austin, TX, 2003.
2. Newsome, J., and D. Song. Dynamic Taint Analysis for Automation Detection, Analysis, and Signature Generation of Exploits on Commodity Software. Proc. of Network and Distributed Security Symposium, San Diego, CA, 2005.
3. Nguyen-Tong, A., et al. Automatically Hardening Web Applications Using Precise Tainting. Proc. of 20th IFIP International Information Security Conference, 2005.
4. Chen, S., J. Xu, N. Nakka, Z. Kalbarczyk, and R.K. Iyer. Defeating Memory Corruption Attacks Via Pointer Taintedness Detection. Proc. of International Conference on Dependable Systems and Networks, 2005: 378-387.
5. Cabuk S., C. Brodley, and C. Shields. IPCovert Timing Channels: Design and Detection. Proc. of the 11th ACM Conference on Computer and Communications Security, 2004: 178-187.
6. Guyer, S.Z., and C. Lin. An Annotation Language for Optimizing Software Libraries. Proc. of the 2nd Conference on Domain-Specific Languages, 1999: 39-52.
7. Denning, D. "A Lattice Model of Secure Information Flow." Communications of the ACM 19.5 (1976): 236-243.
8. Lam, L.C., and T.C. Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. Proc. of the 22nd Annual Computer Security Applications Conference, 2006.
9. Strom, R., and S. Yemini. "Typestate: A Programming Language Concept for Enhancing Software Reliability." IEEE Transactions on Software Engineering 12.1 (1986): 157-171.
10. Guyer, S.Z., and C. Lin. Client-Driven Pointer Analysis. Proc. of the 10th Annual Static Analysis Symposium, June 2003.
11. Newsome, J., D. Brumley, and D. Song. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. Proc. of the Network and Distributed Security Symposium, 2006.
12. Xu, W., S. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. Proc. of the 15th USENIX Security Symposium, 2006.
13. Qin, F., C. Wang, Z. Li, H. Seop Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Information Flow Tracking System for Detecting Security Attacks. Proc. from the 39th Annual IEEE/ACM Symposium on Microarchitecture, 2006: 135-148.

About the Authors

Steve Cook is a senior research analyst in the System Security and High Reliability Software section at the SwRI. His background and expertise are in parallel and real-time computing, compilers, as well as object-oriented and generic programming. Cook received his master's degree in computer science from Texas A&M University. While at Texas A&M, he worked as a research assistant for Dr. Bjarne Stroustrup, creator of the C++ Programming Language, where he helped develop a new approach to writing concurrent programs free from race conditions and deadlock.

SwRI
System Security and
High Reliability Software
P.O. Drawer 28510
San Antonio, TX 78228-0510
Phone: (210) 522-6322
E-mail: steve.cook@swri.org

Calvin Lin, Ph.D., is an associate professor of computer sciences at UT-Austin and director of their Turing Scholars Honors Program. His research takes a broad view at how languages, compilers, and microarchitecture can improve system performance and programmer productivity. Lin leads the development of the Broadway compiler, which exploits domain-specific information in the compilation process. He holds a doctorate in computer science from the University of Washington.

Department of Computer Sciences
UT-Austin
1 University Station C0500
Austin, TX 78712-1188
Phone: (512) 471-9560
E-mail: lin@cs.utexas.edu

Walter Chang received his bachelor's degree in computer science from Cornell University and is currently a doctoral student in the Department of Computer Sciences at UT-Austin, where his research develops program analyses to improve various aspects of software quality, including software security and software correctness.

Department of Computer Sciences
UT-Austin
1 University Station C0500
Austin, TX 78712-1188
Phone: (512) 232-7434
E-mail: walter@cs.utexas.edu

Web Sites

Department of Homeland Security's (DHS) Software Assurance Program

www.us-cert.gov/swa

The DHS Software Assurance Program spearheads the development of practical guidance and tools and promotes research and development of secure software engineering, examining a range of development issues from new methods that avoid basic programming errors to enterprise systems that remain secure when portions of the system software are compromised. Through collaborative software assurance efforts, stakeholders seek to reduce software vulnerabilities, minimize exploitation, and address ways to improve the routine development and deployment of trustworthy software products.

The Open Web Application Security Project (OWASP)

www.owasp.org

The OWASP is a worldwide free and open community focused on improving the security of application software. Their mission is to make application security "visible" so that people and organizations can make informed decisions about application security risks.

SecurityFocus

www.securityfocus.com

SecurityFocus provides the most comprehensive and trusted source of computer and application security information on the Internet. It provides objective, timely, and comprehensive security information to all members of the security community. Information includes computer security vulnerability announcements, security-related news stories and feature articles, effective security measure implementation ideas, and a high volume, full-disclosure mailing list for detailed discussions.

MILS: High-Assurance Security at Affordable Costs

www.cotsjournalonline.com/home/article.php?id=100423&pg=1

This *COTS Journal* article explores how multiple independent levels of security (MILS) build high-assurance systems that must survive high-threat environments. The central idea behind MILS is to partition a system in such a way that 1) the failure or corruption of any single partition cannot affect any other part of the system or network, and 2) each partition can be security-evaluated and certified separately.

Build Security In (BSI)

http://buildsecurityin.us-cert.gov

BSI contains and links to best practices, tools, guidelines, rules, principles, and other resources that software developers, architects, and security practitioners can use to build security into software in every phase of its development. BSI content is based on the principle that software security is fundamentally a software engineering problem and must be addressed in a systematic way throughout the software development life cycle.