FULLY COUNTERING TRUSTING TRUST THROUGH DIVERSE DOUBLE-COMPILING

by

David A. Wheeler

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

_____ Director

_____

_____

_____ Department Chairperson

_____ Program Director (Ph.D. Only)

_____ Dean/Director of College, School or Institute

Date: _____ Fall Semester 2009
George Mason University
Fairfax, VA

Modification date: 2009-07-08

Fully Countering Trusting Trust through Diverse Double-Compiling

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University.

By

David A. Wheeler
Master of Science
George Mason University, 1990

Directors: Dr. Ravi Sandhu and Dr. Daniel Menascé, Professors
Information Technology & Engineering

Fall Semester 2009
George Mason University
Fairfax, Virginia

Dedication

This is dedicated to my extended family, who sacrificed many days so I could perform this work, and to the memory of my former mentors Dennis W. Fife and Donald Macleay, who always believed in me.

Soli Deo gloria—Glory to God alone.

Acknowledgments

I would like to thank my PhD committee members and former members Dr. Ravi Sandhu, Dr. Daniel Menascé, Dr. Paul Ammann, Dr. Jeff Offutt, Dr. Yutao Zhong, and Dr. David Rine, for their helpful comments.

IDA provided a great deal of help. My thanks to IDA management, particularly Dr. Roger Mason and the Honorable Dr. Priscilla Guthrie, who enabled me to work on this and partly supported it through IDA's Central Research Program. I am very grateful to my IDA co-workers (alphabetically by last name) Dr. Brian Cohen, Aaron Hatcher, Dr. Dale Lichtblau, Dr. Reg Meeson, Dr. Clyde Moseberry, Dr. Clyde Roby, Dr. Ed Schneider, Dr. Marty Stytz, and Andy Trice, who had many helpful comments on this paper and/or the previous ACSAC paper. Reg Meeson in particular spent many hours carefully reviewing the proofs and related materials, and I thank him for that. Aaron Hatcher was immensely helpful in working to scale the Diverse Double-Compiling (DDC) technique up to a real-world application using GCC. Aaron helped implement many applications of DDC that we thought should have worked with GCC, but didn't, and then helped to determine *why* they didn't work. These "Edison successes" (which successfully found out what did *not* work) were important in helping to lead to a working application of DDC to GCC. Aaron (along with Reg Meeson) had many helpful comments on the proofs, and was the first to seriously use my graphical notation to describe DDC.

Many others also helped create this work. The seminal work of Paul A. Karger, Roger R. Schell, and Ken Thompson made the world aware of a problem that needed solving; without knowing there was a problem, there would have been no work to solve it. Henry Spencer posted the first version of this idea that eventually led to this paper (though this paper expands on it far beyond the few sentences that he wrote). Henry Spencer, Eric S. Raymond, and the anonymous ACSAC reviewers provided helpful comments on the ACSAC paper. I received many helpful comments and other information after publication of the ACSAC paper, including comments from (alphabetically by last name) Steven M. Bellovin, Terry Bollinger, Ulf Dittmer, Jakub Jelinek, Ben Laurie, Mike Lisanke, Thomas Lord, Bruce Schneier, Brian Snow, and James Walden. Tawnia Wheeler proofread both the ACSAC paper and this document; thank you! My thanks to the many developers of the OpenDocument specification and the OpenOffice.org implementation, who made developing this document a joy.

Table of Contents

List of Tables

List of Figures

List of Abbreviations and Symbols

-A	not A. Equivalent to $\neg A$
A & B	A and B (logical and). Equivalent to $A \wedge B$
A | B	A or B (logical or). Equivalent to $A \vee B$
A -> B	A implies B
ACSAC	Annual Computer Security Applications Conference
aka	also known as
all X A	for all X, A. Equivalent to $\forall X. A$
cA or c_A	Compiler c_A, the compiler under test
cGP or c_{GP}	Compiler c_{GP}, the putative grandparent of c_A
cP or c_P	Compiler P, the putative parent of c_A
CPU	Central Processing Unit
cT or c_T	Compiler c_T, a "trusted" compiler (see paper for definition of "trusted")
DDC	Diverse Double-Compiling
e1, e2	Environments that produce stage1 and stage2
eA, eP	Environments that putatively produced c_A and c_P
eArun	Environment that c_A and stage2 are intended to run in
FOPLE	First-order predicate logic with equality
FSF	Free Software Foundation
GCC	GNU Compiler Collection (formerly GNU C compiler)

GNU	GNU's not Unix
GPL	General Public License
iff	if and only if
OSI	Open Source Initiative
QED	Quod erat demonstrandum ("which was to be demonstrated")
sA or s_A	putative source code of c_A
sP or s_P	putative source code of c_P
tcc or TinyCC	Tiny C Compiler

List of Key Definitions

compiler	An executable that, when executed, translates source code into an executable.
compiling	The process of using a compiler to translate source code into an executable.
corrupted compiler	A corrupted executable that is a compiler.
corrupted executable	An executable that does not correspond to its putative source code (see "corrupted compiler" and "malicious executable").
Diverse Double-Compiling (DDC)	A technique for determining if a compiler is corrupted. First, use a separate "trusted" compiler to compile the source code of the "parent" of the compiler under test. Then, run that resulting executable to compile the purported source code of the compiler under test. Then, check if the final result is exactly identical to the original compiler executable (e.g., bit-for-bit equality) using some trusted means. If it is, then the purported source code and executable of the compiler under test correspond, given some assumptions to be discussed later.
effects	All information or execution timing arising from the environment that can affect the results of a compilation, but is not part of the input source code. This is used to model random number generators, thread execution ordering, differences between platforms allowed by the language, and so on.
environment	A platform that can run executables. This would include the computer hardware (including the central processing unit) and any software that supports or could influence the compiler's result (e.g., the operating system).
executable	Data that can be directly executed by a computing environment. An executable may be code for an actual machine or for a simulated machine (e.g., a "byte code"). A common alternative term for executable is "binary" (e.g., [Sabin2004]), but this term is misleading; in

	modern computers, all data is represented using binary codes. For purposes of this paper, "object code" is a synonym for "executable". Compilers produce executables, and compilers themselves are executables.
malicious compiler	A malicious executable that is a compiler.
malicious executable	A corrupted executable whose corruption was caused by intentional subversion.
maliciously misleading code	Source code that is intentionally designed to look benign, yet creates an vulnerability (including an attack).
payload	Code that actually performs a malicious event (e.g., the inserted malicious code and the code that causes its insertion). These are initiated through triggers.
source code (aka source)	A representation of a program that can be transformed by a compiler into an executable. It is typically human-readable.
subverted compiler	Synonym for "malicious compiler".
trigger	A condition, determined by an attacker, in which a malicious event is to occur (e.g., the condition causing malicious code to be inserted into a program, and the condition that causes the inserted code to take action).
Trojan horse	Software that appears to the user to perform a desirable function but facilitates unauthorized access into the user's computer system.
trusting trust attack	An attack in which an attacker attempts to disseminate a compiler executable that produces corrupted executables, at least one of those corrupted executables is a corrupted compiler, and the attacker attempts to make this situation self-perpetuating.

Abstract

FULLY COUNTERING TRUSTING TRUST THROUGH DIVERSE DOUBLE-COMPILING

David A. Wheeler, M.A.

George Mason University, 2009

Thesis director: Dr. Ravi Sandhu and Dr. Daniel Menascé

An Air Force evaluation of Multics, and Ken Thompson's famous Turing award lecture "Reflections on Trusting Trust," showed that compilers can be subverted to insert malicious Trojan horses into critical software, including themselves. If this "trusting trust" attack goes undetected, even complete analysis of a system's source code will not find the malicious code that is running. Previously-known countermeasures have been grossly inadequate. If this attack cannot be countered, attackers can quietly subvert entire classes of computer systems, gaining complete control over financial, infrastructure, military, and/or business system infrastructures worldwide. This dissertation's thesis is that the trusting trust attack can be detected and effectively countered using the "Diverse Double-Compiling" (DDC) technique, as demonstrated by (1) a formal proof that DDC can determine if source code and generated executable code correspond, (2) a demonstration of DDC with three compilers (a small C compiler, a small malicious Lisp

compiler, and a large industrial-strength C compiler, GCC), and (3) a description of approaches for applying DDC in various real-world scenarios, including a description of how to scale DDC up to entire operating systems. In the DDC technique, source code is compiled twice: once with a second (trusted) compiler (using the source code of the compiler's parent), and then the compiler source code is compiled using the result of the first compilation. If the result is bit-for-bit identical with the untrusted executable, then the source code accurately represents the executable.

1 Introduction

Many software security evaluations examine source code, under the assumption that a program's source code accurately represents the executable actually run by the computer[1]. Naïve developers presume that this can be assured simply by recompiling the source code to see if the same executable is produced. Unfortunately, the "trusting trust" attack can falsify this presumption.

For purposes of this paper, an executable that does not correspond to its putative source code is *corrupted*. If a corrupted executable was intentionally created, we can call it a *malicious* executable. The *trusting trust attack* occurs when an attacker attempts to disseminate a compiler executable that produces corrupted executables, at least one of those corrupted executables is a corrupted compiler, and the attacker attempts to make this situation self-perpetuating. The attacker may use this attack to insert other Trojan horse(s) (software that appears to the user to perform a desirable function but facilitates unauthorized access into the user's computer system).

[1]An *executable* is data that can be directly executed by a computing environment. An executable may be code for an actual machine or for a simulated machine (e.g., a "byte code"). An common alternative term for executable is "binary" (e.g., [Sabin2004]), but this term is misleading; in modern computers, *all* data is represented using binary codes. For purposes of this paper, "object code" is a synonym for "executable". *Source code* is a representation of a program that can be translated into an executable, and is typically human-readable. A *compiler* is an executable that when executed translates source code into an executable. The process of using a compiler to translate source code into an executable is termed *compiling*.

Information about the trusting trust attack was first published in [Karger1974]; it became widely known through [Thompson1984]. Unfortunately, there has been no practical way to fully detect or counter the trusting trust attack, because repeated in-depth review of industrial compilers' executable code is impractical.

For source code evaluations to be strongly credible, there must be a way to justify that the source code being examined accurately represents what is being executed—yet the trusting trust attack subverts that very claim. Internet Security System's David Maynor argues that the risk of attacks on compilation processes is increasing [Maynor2004] [Maynor2005]. Karger and Schell noted that the trusting trust attack was still a problem in 2000 [Karger2000], and some technologists doubt that computer-based systems can ever be secure because of the existence of this attack [gauis2000]. Anderson et al. argue that the general risk of subversion is increasing [Anderson2004].

Recently, in several mailing lists and blogs, a special technique to detect such attacks has been briefly described, which uses a second (diverse) "trusted" compiler and two compilation stages. This paper terms the technique "diverse double-compiling" (DDC). In DDC, if the final result is bit-for-bit identical to the original compiler executable and certain other assumptions hold, then the compiler executable corresponds with its source code. However, this work began, there had been no examination of DDC in detail which identified its assumptions, proved its correctness or effectiveness, or discussed practical issues in applying it. There had also not been any public demonstration of DDC.

This dissertation's thesis is that the trusting trust attack can be detected and effectively countered using the "Diverse Double-Compiling" (DDC) technique, as demonstrated by (1) a formal proof that DDC can determine if source code and generated executable code correspond, (2) a demonstration of DDC with three compilers (a small C compiler, a small malicious Lisp compiler, and a large industrial-strength C compiler, GCC), and (3) a description of approaches for applying DDC in various real-world scenarios, including a description of how to scale DDC up to entire operating systems.

This paper provides background and a description of the threat, followed by an informal description of DDC. This is followed by a formal proof of DDC, information on how diversity (a key requirement of DDC) can be increased, three demonstrations of DDC, and information on how to overcome practical challenges in applying DDC. The paper closes with a summary and ramifications. Appendices have some additional detail; further details, including materials sufficient to reproduce the experiments, are available at <http://www.dwheeler.com/trusting-trust/dissertation>.

This dissertation uses logical (British) quoting conventions; quotes do not enclose punctuation unless they are part of the quote [Ritter2002]. Including extraneous characters in a quotation can be grossly misleading, especially in computer-related material [Raymond2003, chapter 5].

2 Background and related work

This section provides background and related work. It begins with a discussion of the initial revelation of the trusting trust attack by Karger, Schell, and Thompson, including a brief description of "obvious" yet inadequate solutions. The next subsections discuss work on corrupted or subverted compilers, general work on analyzing software, and general approaches for using diversity to improve security. This is followed by evidence that software subversion is a real problem, not just a theoretical concern. This section concludes by discussing the DDC paper published by the Annual Computer Security Applications Conference (ACSAC) [Wheeler2005] and the improvements to DDC that have been made since that time.

2.1 Initial revelation: Karger, Schell, and Thompson

Karger and Schell provided the first public description of the problem that compiler executables can insert malicious code into themselves. They noted in their examination of Multics vulnerabilities that a "penetrator could insert a trap door into the... compiler... [and] since the PL/I compiler is itself written in PL/I, the trap door can maintain itself, even when the compiler is recompiled. Compiler trap doors are significantly more complex than the other trap doors... However, they are quite practical to implement" [Karger1974].

Ken Thompson widely publicized this problem in his famous 1984 Turing Award presentation “Reflections on Trusting Trust”, clearly explaining it and demonstrating that this was both a practical and dangerous attack. He described how to modify the Unix C compiler to inject a Trojan horse, in this case to modify the operating system login program to surreptitiously give him root access. He also added code so that the compiler would inject a Trojan Horse when compiling itself, so the compiler became a “self-reproducing program that inserts both Trojan horses into the compiler”. Once this is done, the attacks could be removed from the source code. At that point no source code examination—even of the compiler—would reveal the existence of the Trojan horses, yet the attacks could persist through recompilations and cross-compilations of the compiler. He then stated that “No amount of source-level verification or scrutiny will protect you from using untrusted code... I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these defects will be harder and harder to detect” [Thompson1984]. Thompson’s demonstration subverted the login program (for control) and the disassembler (to hide the attack from disassembly). As a demonstration, Thompson implemented this attack in the C compiler and successfully subverted another Bell Labs group; the attack was never detected.

Thompson later gave more details about his demonstration, including assurances that the malicious compiler was never released outside Bell Labs [Thornburg2000].

Obviously, this attack invalidates security evaluations based on source code review, and recompilation of source code using a potentially-corrupted compiler does not eliminate the problem. Some simple approaches appear to solve the problem at first glance, yet fail to do so or have have significant weaknesses:

1. Compiler executables could be manually compared with their source code. This is impractical given compilers' large sizes, complexity, and rate of change.
2. Such comparison could be automated, but optimizing compilers make such comparisons extremely difficult, compiler changes make keeping such tools up-to-date difficult, and the tool's complexity would be similar to a compiler's.
3. A second compiler could compile the source code, and then the executables could be compared automatically to argue semantic equivalence. There is some work in determining the semantic equivalence of two different executables [Sabin2004], but this is very difficult.
4. Receivers could require that they only receive source code and then recompile everything themselves. This fails if the receiver's compiler is already malicious; thus, it simply moves the attack location. An attacker could also insert the attack into the compiler's source; if the receiver accepts it (due to lack of diligence or conspiracy), the attacker could remove the evidence in a later version.
5. Programs can be written in interpreted languages. But eventually an interpreter must be implemented by machine code, so this simply moves the attack location.

2.2 Other work on corrupted compilers

Some previous papers outline approaches for countering corrupted compilers, though their approaches have significant weaknesses. Draper [Draper1984] recommends screening out malicious compilers by writing a "paraphrase" compiler (possibly with a few dummy statements) or a different compiler executable, compiling once to remove the Trojan horse, and then compiling a second time to produce a Trojan horse-free compiler. This idea is expanded upon by McDermott [McDermott1988], who notes that the alternative compiler could be a reduced-function compiler or one with large amounts of code unrelated to compilation. Lee's "approach #2" describes most of the basic process of diverse double-compiling, but implies that the results might not be bit-for-bit identical. [Lee2000]. Luzar makes a similar point as Lee, describing how to rebuild a system from scratch using a different trusted compiler but not noting that the final result should be bit-for-bit identical if other factors are carefully controlled [Luzar2003].

None of these papers note that it is possible to produce a result that is bit-for-bit identical to the original compiler executable. This is a significant advantage of diverse double-compiling (DDC), because determining if two different executables are "functionally equivalent" is extremely difficult[2], while determining if two executables are identical is extremely easy. These previous approaches require each defender to recompile their compiler themselves before using it; in contrast, DDC can be used as an after-the-fact vetting by multiple third parties, without requiring a significant change in compiler delivery or installation processes, and without requiring that all compiler users receive

[2]Determining if two executables are equivalent is undecidable in general; see section 5.6.1.

the compiler source code. All of these previous approaches simply move the potential vulnerability somewhere else (e.g., to the process using the "paraphrase" compiler); in contrast, an attacker who wishes to avoid detection by DDC must corrupt *both* the original compiler and *every* application of DDC to that executable, so every application of DDC further builds trust that a specific executable corresponds with its putative source code. Also, none of these papers demonstrate their technique.

Magdsick discusses using different versions of a compiler and different compiler platforms (CPU and operating system) to check executables, but presumes that the compiler itself will simply be the same compiler (just a different version). He does note the value of recompiling "everything" to check it [Magdsick2003]. Anderson notes that cross-compilation does not help if the attack is in the compiler [Anderson2003]. Mohring argues for the use of recompilation by GCC to check other components, presuming that the GCC executables themselves in some environments would be pristine [Mohring2004]. He makes no notice that all GCC implementations used might be malicious, or of the importance of diversity in compiler implementation. In his approach different compiler versions may be used, so outputs would be "similar" but not identical; this leaves the difficult problem of comparing executables for "exact equivalence" unresolved.

Much work has been done to develop proofs of correctness for compilers, either of the compiler itself and/or its generated results [Dave2003] [Stringer-Calvert1998] [Bellovin1982]. This is quite difficult even for simple languages, though there has been

progress. [Leinenbach2005] discusses progress in verifying a subset C compiler using Isabelle/HOL. "Compcert" is a compiler that generates PowerPC assembly code from Clight (a large subset of the C programming language); this compiler is primarily written using the specification language of the Coq proof assistant, and its correctness (that the generated assembly code is semantically equivalent to its source program) has been entirely proved within the Coq proof assistant [Leroy2006] [Blazy2006] [Leroy2008] [Leroy2009]. [Goerigk1997] requires formal specifications and correspondence proofs, along with double-checking of resulting transformations with the formal specifications. It does briefly note that "if an independent (whatever that is) implementation of the specification will generate an equal bootstrapping result, this fact might perhaps increase confidence. Note however, that, in particular in the area of security... We want to guarantee the correctness of the generated code, e.g., preventing criminal attacks" [Goerigk1997, 17]. However, it does not explain what independence would mean, nor what kind of confidence this equality would provide. [Goerigk1999] specifically focuses on countering Trojan horses in compilers, through formal verification techniques, but again this requires having formal specifications and performing formal correspondence proofs. Goerigk recommends "a posteriori code inspection based on syntactic code comparison" to counter the the trusting trust attack, but such inspection is very labor-intensive on industrial-scale compilers that implement significant optimizations. DDC can be dramatically strengthened by having formal specifications and proofs of compilers (which can then be used as the trusted compiler), but DDC does not require them. Indeed, DDC and formal proofs of compilers can be used in a complementary way: A

formally-proved compiler may omit many useful optimizations (as they can be difficult or time-consuming to prove), but it can still be used as the DDC "trusted compiler" to gain confidence in another (production-ready) compiler.

Spinellis argues that "Thompson showed us that one cannot trust an application's security policy by examining its source code... The recent Xbox attack demonstrated that one cannot trust a platform's security policy if the applications running on it cannot be trusted." [Spinellis2003]. It is worth noting that the literature for change detection (such as [Kim1994] and [Forrest1994]) and intrusion detection do not easily address this problem. Here the compiler is operating normally: it is expected to accept source code and generate object code.

Faigon [Faigon]'s "Constrained Random Testing" detects compiler defects by creating many random test programs, compiling them with a compiler under test and a reference compiler, and detecting if running them produces different results. Faigon's approach may be useful for finding some compiler errors, but it is extremely unlikely to find the kind of malicious compilers as considered here.

A common test for errors used by many compilers (including GCC) is the so-called "compiler bootstrap test". Goerigk formally describes this test, crediting Niklaus Wirth's 1986 book *Compilerbau* as proposing this test for detecting errors in compilers [Goerigk1999]. In this test, if c(s,b) is the result of compiling source s using compiler executable b, and $\overline{m}$ is some other compiler (the "bootstrap" compiler), then[3]:

[3]This is theorem 2 (the bootstrap test theorem) of [Goerigk1999]. For clarity, the text has been modified so that its notation is the same as the notation used in this dissertation.

> *If m0 and s are both correct and deterministic, $\overline{m}$ is correct, m0=c(s,$\overline{m}$), m1=c(s,m0), m2=c(s,m1), all compilations terminate, and if the underlying hardware works correctly, then m1=m2.*

The compiler bootstrap test goes through steps to determine if m1=m2; if not, there is a compiler error of some kind. This test finds many unintentional errors, which is why it is popular. But [Goerigk1999] points out that this test is insufficient to make strong claims, in particular, m1 may equal m2 even if $\overline{m}$, m0, or s are *not* correct. For example, it is trivial to create compiler source code that passes this test, yet is incorrect, since this test only tests features used in the compiler itself. More importantly, for purposes of this paper, if $\overline{m}$ is a malicious compiler, this test can pass yet produce a malicious compiler m2 (all that is required is that $\overline{m}$ have triggers and payloads for the compiler described in source code s). Using a bootstrap compiler $\overline{m}$ that's less likely to have such triggers and payloads is the essence of Draper's approach ([Draper1984]), as discussed earlier. Note that the compiler bootstrap test does *not* consider the possibility of using two different bootstrap compilers ($\overline{m}'$ and $\overline{m}''$), and later comparing their different m2 compiler results (m2′ and m2″) to see if they produce the same (bit-for-bit) result. Therefore, the DDC technique is *not* the same as the compiler bootstrap test. However, DDC *does* have many of the same preconditions as the compiler bootstrap test. Since the compiler bootstrap test is popular, many DDC preconditions are already met by typical industrial compilers, making DDC easier to apply to typical industrial compilers.

2.3 Analyzing software

All programs can be analyzed to find intentionally-inserted or unintentional security issues (aka vulnerabilities). These techniques can be broadly divided into static analysis (which examines a static representation of the program, such as source code or executable, without running it) and dynamic analysis (which examines what the program does while it is executing). Formal methods, which are techniques that use mathematics to prove programs or program models are correct, are a specific kind of static analysis technique. Some analysis processes can combine these techniques. In many cases, their application is heavily affected by the programming language (e.g., [Younan2004] provides a survey of static and dynamic countermeasures for C/C++ code injection).

Since compilers are programs, these general analysis techniques (both static and dynamic) that are not specific to compilers can be used on compilers as well.

2.3.1 Static analysis

Static analysis techniques examine programs (their source code, executable, or both) without executing them. Both programs and humans can perform static analysis.

There are many static analysis programs (aka tools) available; many are focused on identifying security vulnerabilities in software. The NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project (http://samate.nist.gov) is "developing methods to enable software tool evaluations, measuring the effectiveness of tools and techniques, and identifying gaps in tools and methods". SAMATE has collected a long list of static

analysis programs for finding security vulnerabilities by examining source code or executable code. There are also a number of published reports comparing various static analysis tools, such as [Zitser2004], [Forristal2005], [Kratkiewicz2005], and [Michaud2006]. A draft functional specification for source code analysis tools has been developed [Kass2006], proposing a set of defects that such tools would be required to find and the code complexity that they must be able to handle while detecting them.

Although [Kass2006] notes that source code analysis tools can in theory find malicious trap doors, many documents on static analysis emphasize finding *unintentional* errors, not maliciously-implanted vulnerabilities. As with the tools it is designed to specify, [Kass2006] specifies searches for a specific set of errors that have been made many times in real programs, and limits the required depth of the analysis (to make analysis time and reporting manageable). [Chou2006] also notes that in practice, static analyzers give up on error classes that are too hard to diagnose. For unintentional vulnerabilities, this is sensible; errors that have commonly occurred before are likely to recur again (so searching for them can be very helpful). Vulnerabilities that are difficult for tools to find are probably also more difficult for attackers to find, and developers may be wiser if they first fix the defects that are easier to find. But these approaches are less helpful against a malicious adversary who is *inserting* specialized malicious code into a program. An adversary could intentionally insert one of these common errors, perhaps because they have high deniability, but ensure that the code complexity is sufficiently high that the tool will not find it. Alternatively, an adversary could simply insert code that is an attack but not in the list of patterns the tools search for. Indeed, an adversary can use those static

analysis tools to determine that the malicious code will *not* be detected later. [Kass2006] discusses checking for hard-coded passwords, but not for triggering an action on a date or time, so tools complying with this specification would not catch most of the subversions listed above. Indeed, the list of actions a program "shouldn't" do is infinite.

Static analysis tools also exist for analyzing executable files, instead of source code files. Indeed, [Balakrishnan2005] argues that program analysis should begin with executables instead of source code, because only the executables are actually run and source code analysis can be misled. To address this, there are efforts to compute better higher-level constructs from executable code, but in the general case this is still a difficult research area [Linger2006].

[Wysopal] presents a number of heuristics that can be used to statically detect some application backdoors in executable files. This includes identifying static variables that "look like" usernames, passwords, or cryptographic keys, searching for network API calls in applications where they are unexpected, searching for standard date/time API calls (which may lead to a time bomb), and so on. Unfortunately, many malicious programs will not be detected by such heuristics, and attackers can develop malicious software in ways that avoid detection by such heuristics.

Many static analysis tools for executables use the same basic approach as most static analysis tools for source code: they search for specific programs or program fragments known to be problematic. The most obvious case are virus-checkers; though it is possible to examine behavior, and some anti-virus programs are increasingly doing so,

traditionally “anti-virus” programs have a set of patterns of known viruses, which is constantly updated and used to search various executables (e.g., file or boot record) to see if these patterns are present [Singh2002] [Lapell2006]. However, as noted in Fred Cohen’s original seminal work on viruses [Cohen1985], viruses can mutate as they propagate, and it is not possible to create a pattern listing all-and-only malicious programs. [Christodorescu2003] attempts partially counter this; this paper regards malicious code detection as an obfuscation-deobfuscation game between malicious code writers and researchers, and presents an architecture for detecting known malicious patterns in executables that are hidden by common obfuscation techniques. Even this more robust architecture does not work against different malicious patterns, nor against different obfuscation techniques.

Of course, even if tools cannot find malicious code, detailed human review can be used at the source or executable level if the software is critical enough to warrant it. For example, the OpenBSD operating system source code is regularly and purposefully examined by a team of people with the explicit intention of finding and fixing security holes, and as a result has an excellent security record [Payne2002]. The Strategic Defense Initiative Organization (SDIO) even developed a set of process requirements to counter malicious and unintentional vulnerabilities, emphasizing multi-person knowledge and review along with configuration management and other safeguards [SDIO1993].

Unfortunately, the trusting trust attack can render human reviews moot if there is no technique to counter it. The trusting trust attack immediately renders examination of the

source code inadequate, because the executable code need not correspond to the source code. Thompson's attack subverted the symbolic debugger, so in that case, even human review of the executable would fail to detect the attack. Thus, human reviews are less convincing unless the trusting trust attack is itself countered.

Human review also presumes that other humans examining source code or executables will be able to detect malicious code. In large code bases, this can be a challenge simply due to their size and complexity. In addition, it is possible for an adversary to create source code that *appears* to work correctly, yet actually performs a malevolent action instead. This paper uses the term *maliciously misleading code* for any source code that is intentionally designed to look benign, yet creates an vulnerability (including an attack). The topic of maliciously misleading code is further discussed in section 8.11.

2.3.2 Dynamic analysis

It is also possible to use dynamic techniques in an attempt to detect and/or counter vulnerabilities by examining the activities of a system, and then halting or examining the system when those activities are suspicious. A trivial example is execution testing, where a small set of inputs are provided and the inputs are checked to see if they are correct. However, dynamic analysis is completely inadequate for countering the trusting trust attack.

Traditional execution testing is unlikely to counter the trusting trust attack. Such attacks will only "trigger" on very specific inputs, as discussed in section 3.2, so even if the

executable is examined in detail, it is extremely unlikely that traditional execution testing will detect this problem.

Detecting at run-time arbitrary corrupted code in a compiler or the executable code it generates is very difficult. The fundamental behavior of a corrupted compiler – that it accepts source code and generates an executable – is no different from a uncorrupted one, making dynamic techniques *by themselves* difficult to apply. Similarly, any malicious code a compiler inserts into other programs can often be made to appear as normal behavior. For example, a login program with a trap door (a hidden username and password) has the same general behavior: It decides if a user may log in and what privileges to apply.

In theory, continuous comparison of every dynamic execution to its source code could detect differences between the executable and source code. Unfortunately, this would need to be done all the time, draining performance. Even worse, tools to do this comparison, given modern compilers producing highly optimized code, would be far more complex than a compiler.

Given an extremely broad definition of "system", the use of software configuration management tools and change detection tools like Tripwire [Kim1994] could be considered dynamic techniques for countering malicious software. Both enable detection of changes in the behavior of a larger system. Certainly a configuration management system could be used to record changes made to compiler source, and then used to enable reviewers to examine just the differences. But again, such review presupposes that any

vulnerability in an executable could be revealed by analyzing its source code, a presupposition the trusting trust attack subverts.

A broader problem is that once code is running, *some* programs must be trusted, and at least some of that code will almost certainly have been generated by a compiler. Any program that attempts to monitor execution might itself be subverted, just as Thompson subverted the symbolic debugger, unless there is a technique to prevent it. In any case, it would be better to detect and counter malicious code *before* it executed, instead of trying to detect malicious code's execution while or after it occurs.

2.4 Diversity for security

There are a number of papers and articles about employing diversity to aid computer security, though they generally do not discuss or examine how to use diversity to counter Trojan horses inside compilers themselves or the compilation environment.

Geer et al. strongly argue that a monoculture (an absence of diversity) in computing platforms is a serious security problem [Geer2003] [Bridis2003], but do not discuss employing compiler diversity to counter this particular attack.

Forrest et al [Forrest1997] argues that run-time diversity in general is beneficial for computer security. In particular, their paper discusses techniques to vary final executables by "randomized" transformations affecting compilation, loading, and/or execution. Their goal was to automatically change the executable (as seen at run-time) in some random ways sufficient to make it more difficult to attack. The paper provides a set

of examples, including adding/deleting nonfunctional code, reordering code, and varying memory layout. They demonstrated the concept through a compiler that randomized the amount of memory allocated on a stack frame, and showed that the approach foiled a simple buffer overflow attack. Again, they do not attempt to counter corrupted compilers.

2.5 Subversion of software is a real problem

Subversion of software is not just a theoretical possibility; it is a current problem. One book on computer crime lists various kinds of software subversion as attack methods (e.g., trap doors, Trojan horses, viruses, worms, salamis, and logic bombs) [Icove1995, 57-58]. Examples of specific software subversion or subversion attempts include:

- Michael Lauffenburger inserted a logic bomb into a program at defense contractor General Dynamics, his employer. The bomb would have deleted vital rocket project data in 1991, including much that was unrecoverable, but another employee stumbled onto it before it was triggered [AP1991] [Hoffman1991].
- Timothy Lloyd planted a 6-line logic bomb into the systems of Omega Engineering, his employer, that went off on July 31, 1996. This erased all of the company's contracts and proprietary software used by their manufacturing tools, resulting in an estimated $12 million in damages, 80 people permanently losing their jobs, and the loss of their competitive edge in the electronics market space. Plant manager Jim Ferguson stated flatly, "We will never recover." On Feb. 26, 2002, a judge sentenced Lloyd to 41 months in prison, three years of probation,

and ordered him to pay more than $2 million in damages to Omega [Ulsh2000] [Gardian].

- Roger Duronio worked at UBS PaineWebber's offices in Weehawken, N.J., and "was with the company for two years while he served as a system administrator." Apparently dissatisfied with his pay, he installed a logic bomb to detonate on March 4, 2002, and resigned from the company. When the logic bomb went off, it caused over 1,000 of their 1,500 networked computers to begin deleting files. This cost UBS PaineWebber more than $3 million to assess and repair the damage, plus an undetermined amount from lost business. Duronio was sentenced to 97 months in federal prison (the maximum per the U.S. sentencing guidelines), and ordered to make $3.1 million in restitution. [DoJ2006] [Gaudin2006b] The attack was only a few lines of C code, which examined the time to see if it was the detonation time, and then (if so) executed a shell command to erase everything [Gaudin2006a].
- An unnamed developer inside Borland inserted a back door into the Borland/Inprise Interbase SQL database server around 1994. This was a "superuser" account ("politically") with a known password ("correct"), which could not be "changed using normal operational commands, nor [deleted] from existing vulnerable servers." Versions released to the public from 1994 through 2001 included this back door. Originally Interbase was a proprietary program sold by Borland/Inprise. However, it was released as open source software[4] in

[4]Open source software is, briefly, software where users have the right to use the software for any purpose, review it, modify, and redistribute it (modified or not) without requiring royalty payments [Wheeler2007]. The Open Source Definition [OSI2006] and the Free Software Definition [FSF2009] have

July 2000, and less than six months later the open source software developers discovered the vulnerability [Havrilla2001a] [Havrilla2001b]. The Firebird project, an alternate open source software package based on the same Interbase code, was also affected. Jim Starkey, who launched InterBase but left in 1991 before the back door was added to the software in 1994, stated that he believed that this back door was not malicious, but simply added to enable one part of the database software to communicate with another part [Shankland2001]. However, this code had the hallmarks of many malicious back doors: It added a special account that was (1) undocumented, (2) cannot be changed, and (3) gave complete control to the requester.

- An unknown attacker attempted to insert a malicious back door in the Linux kernel in 2003. The two new lines were crafted to *appear* legitimate, by using an "=" where a "==" would be expected. The configuration management tools immediately identified a discrepancy, and examination of the changes by the Linux developers quickly determined that it was an attempted attack [Miller2003] [Andrews2003].

Many have noted insertion of malicious code into software as an important risk:

- Many have noted subversion of software as an issue in electronic voting machines [Saltman1988] [Kohno2004] [Feldman2006] [Barr2007].

---

more formal definitions for this term or the related term "Free software". There is quantitative data showing that, in many cases, using open source software / Free software (abbreviated as OSS/FS, FLOSS, or FOSS) is a reasonable or even superior approach to using their proprietary competition according to various measures [Wheeler2007]. In almost all cases, it is commercial software [Wheeler2009].

- The U.S. Department of Defense (DoD) established a "software assurance initiative" in 2003 to examine software assurance issues in defense software, including how to counter intentionally inserted malicious code [Komaroff2005]. In 2004, the U.S. General Accounting Office (GAO) criticized the DoD, claiming that the DoD "policies do not fully address the risk of using foreign suppliers to develop weapon system software... policies [fail to focus] on insider threats, such as the insertion of malicious code by software developers..." [GAO2004]. The U.S. Committee on National Security Systems (CNSS) defines Software Assurance (SwA) as "the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at anytime during its lifecycle, and that the software functions in the intended manner" [CNSS2006]. Note that intentionally-created vulnerabilities inserting during software development are specifically included in this definition.
- The President's Information Technology Advisory Committee (PITAC) found that "Vulnerabilities in software that are introduced by mistake or poor practices are a serious problem today. In the future, the Nation may face an even more challenging problem as adversaries – both foreign and domestic – become increasingly sophisticated in their ability to insert malicious code into critical software" [PITAC2005, 9]. The U.S. National Strategy to Secure Cyberspace reported that a "spectrum of malicious actors can and do conduct attacks against our critical information infrastructures. Of primary concern is the threat of organized cyber attacks capable of causing debilitating disruption to our Nation's

critical infrastructures, economy, or national security.... [and could lace] our infrastructure with back doors and other means of access." [PCIB2003,6]

- In 2003, China's State Council announced a plan requiring all government ministries to buy only locally produced software when upgrading, and to increase use of open source software, in part due to concerns over "data spyholes installed by foreign powers" in software they procured for government use [CNETAsia2003].

In short, as software becomes more pervasive, subversion of it becomes ever more tempting to powerful individuals and institutions. Attackers can even buy legitimate software companies, or build them up, to widely disseminate quality products at a low price... but with "a ticking time bomb inside." [Schwartau1994, 304-305]

Not all articles about subversion specifically note the trusting trust attack as an issue, but as noted earlier, for source code evaluations to be strongly credible, there must be a way to justify that the source code being examined accurately represents what is being executed—yet the trusting trust attack subverts that very claim. Internet Security System's David Maynor argues that the risk of attacks on compilation processes is increasing [Maynor2004] [Maynor2005]; Karger and Schell noted that the trusting trust attack was still a problem in 2000 [Karger2000], and some technologists doubt that computer-based systems can ever be secure because of the existence of this attack [gauis2000]. Anderson et al. argue that the general risk of subversion is increasing [Anderson2004].

2.6 Previous DDC paper

Initial results from this research in DDC were published by the Annual Computer Security Applications Conference (ACSAC) in [Wheeler2005]. This paper was well-received, for example, Bruce Schneier wrote a glowing review and summary of the paper [Schneier2006], and the Spring 2006 class "Secure Software Engineering Seminar" of Dr. James Walden (Northern Kentucky University) included it in its required reading list.

This dissertation includes the results of [Wheeler2005] and refines it further:

1. The definition of DDC is generalized to cover the case where the compiler is not self-regenerating. Instead, a compiler under test may have been generated using a different "parent" compiler. Self-regeneration (where the putative source code of the parent and compiler under test are the same) is now a special case.
2. A formal proof of DDC is provided, including a formalization of DDC assumptions. The earlier paper includes only an informal justification. The proof covers cases where the environments are different, including the effect of different character representation systems.
3. A demonstration of DDC with a known malicious compiler is shown. As expected, DDC detects this case.
4. A demonstration of DDC with an industrial-strength compiler (GCC) is shown.
5. The discussion on the application of DDC is extended to cover additional challenges, including its potential application to hardware.

3 Description of threat

Thompson describes how to perform the trusting trust attack, but there are some important characteristics of the attack that are not immediately obvious from his presentation. This section examines the threat in more detail and introduces terminology to describe the threat. This terminology will be used later to explain how the threat is countered. For a more detailed model of this threat, see [Goerigk2000] and [Goerigk2002] which provide a formal model of the trusting trust attack (using ACL2).

The following subsections describe what might motivate an attacker to actually perform such an attack, and the mechanisms an attacker uses that make this attack work (triggers, payloads, and non-discovery).

3.1 Attacker motivation

Understanding any potential threat involves determining the benefits to an attacker of an attack, and comparing them to the attacker's risks, costs, and difficulties. Although this trusting trust attack may seem exotic, its large benefits may outweigh its costs to some attackers.

The potential benefits are immense to a malicious attacker. A successful attacker can completely control all systems that are compiled by that executable and that executable's

descendants, e.g., they can have backdoor passwords inserted for logins and gain unlimited privileges on entire classes of systems. Since detailed source code reviews will not find the attack, even defenders who have highly valuable resources and check all source code are vulnerable to this attack.

For a widely-used compiler, or one used to compile a widely-used program or operating system, this attack could result in global control. Control over banking systems, financial markets, militaries, or governments could be gained with a single attack. An attacker could possibly acquire limitless funds (by manipulating the entire financial system), acquire or change extremely sensitive information, or disable a nation's critical infrastructure on command.

An attacker can perform the attack against multiple compilers as well. Once control is gained over all systems that use one compiler, trust relationships and network interconnections could be exploited to ease attack against other compiler executables. This would be especially true of a patient and careful attacker; once a compiler is subverted, it is likely to stay subverted for a long time, giving an attacker time to use it to launch further attacks.

An attacker (either an individual or an organization) who subverted a few of the most widely used compilers of the most widely-used operating systems could effectively control, directly or indirectly, almost every computer in existence.

The attack requires knowledge about compilers, effort to create the attack, and access (gained somehow) to the compiler executable, but all are achievable. Compiler construction techniques are standard Computer Science course material. The attack requires the insertion of relatively small amounts of code, so the attack can be developed by a single knowledgeable person in their spare time. Access rights to change the relevant compiler executables might be harder to acquire, but there are clearly some who have such privileges already, and a determined attacker could acquire such privileges through a variety of means (including network attack, social engineering, physical attack, bribery, and betrayal).

The amount of power this attack offers is great, so it is easy to imagine a single person deciding to perform this attack for their own ends. Individuals entrusted with compiler development might even succumb to the temptation if they believed they could not be caught, and the legion of current virus writers shows that people are willing to write malicious code even without gaining the control this attack can provide.

Given such extraordinarily large benefits to an attacker, a highly resourced organization (such as a government) might decide to undertake it. Such an organization could supply hundreds of experts, working together full-time to deploy attacks over a period of decades. Defending against this scale of attack is far beyond the defensive abilities of most companies and non-profit organizations who develop and maintain popular compilers.

In short, this is an attack that can yield complete control over a vast number of systems, even those systems whose defenders perform independent source code analysis (e.g., those who have especially high-value assets), so it is worth defending against.

3.2 Triggers, payloads, and non-discovery

The trusting trust attack depends on three things: triggers, payloads, and non-discovery. For purposes of this paper, a "trigger" is a condition determined by an attacker in which a malicious event is to occur (e.g., malicious code is inserted into a program). A "payload" is the code that actually performs the malicious event (e.g., the inserted malicious code and the code that causes its insertion). By "non-discovery," this paper means that victims cannot determine if a executable has been tampered with in this way; the lack of transparency in executable files makes this attack possible.

For this attack to be valuable, there must be at least two triggers: one to cause a malicious attack directly of value to the attacker (e.g., detecting compilation of a "login" program so that a Trojan horse can be inserted into it), and another to propagate attacks into future versions of the compiler.

If a trigger is activated when the attacker does not intend the trigger to be activated, the probability of detection increases. However, if a trigger is not activated when the attacker intends it to be activated, then that particular attack will be disabled. If all the attacks by the compiler against itself are disabled, then the attack will no longer propagate; once the compiler is recompiled, the attacks will disappear. Similarly, if a

payload requires a situation that (through the process of change) disappears, then the payload will no longer be effective (and its failure may reveal the attack).

In this paper, "fragility" is the susceptibility of this attack to failure, i.e., that a trigger will activate when the attacker did not wish it to (risking a revelation of the attack), fail to trigger when the attacker would wish it to, or that the payload may fail to work as intended. Fragility is unfortunately less helpful to the defender than it might first appear. An attacker can counter fragility by simply incorporating many narrowly-defined triggers and payloads. Even if a change causes one trigger to fail, another trigger may still fire. By using multiple triggers and payloads, an attacker can attack multiple points in the compiler and attack different subsystems as final targets (e.g., the login system, the networking interface, and so on). Thus, there may be enough vulnerabilities in the resulting system to allow attackers to re-enter and re-insert new triggers and payloads into a malicious compiler. Even if a compiler misbehaves from malfunctioning malware, the results could appear to be a mysterious compiler defect; if programmers "code around" the problem, the attack will stay undetected.

Since attackers do not want their malicious code to be discovered, they may limit the number of triggers/payloads they insert and the number of attacked compilers. In particular, attackers may tend to attack only "important" compilers (e.g., compilers that are widely-used or used for high-asset projects), since each compiler they attack (initially or to add new triggers and payloads) increases the risk of discovery. However, since these attacks can allow an attacker to deeply penetrate systems generated with the

compiler, malicious compilers make it easier for an attacker to re-enter a previously penetrated development environment to refresh an executable with new triggers and payloads. Thus, once a compiler has been subverted, it may be difficult to undo the damage without a process for ensuring that there are no attacks left.

The text above might give the impression that only the compiler itself can influence results (or how they are run), yet this is obviously not true. Assemblers and loaders are excellent places to place a trigger (the popular GCC C compiler actually generates assembly language as text and then invokes an assembler). An attacker could place the trigger mechanism in the compiler's supporting infrastructure such as the operating system kernel, libraries, or privileged programs. In many cases writing triggers is more difficult for such components, but in some cases (such as I/O libraries) this is fairly easy to do.

4 Informal description of Diverse Double-Compiling (DDC)

The idea of diverse double-compiling (DDC) was first created and posted by Henry Spencer in 1998 [Spencer1998] in a very short posting. It was inspired by McKeeman et al's exercise for detecting compiler defects [McKeeman1970] [Spencer2005]. Since this time, this idea has been posted in several places, typically with very short descriptions [Mohring2004] [Libra2004] [Buck2004]. The following subsection describes the graphical notation for describing DDC that is used in this paper. This is followed by a brief informal description of DDC (in its full generality), an informal discussion of its assumptions, and a discussion of a common special case: Self-parenting compilers. This section closes by answering a common question: Why not *always* use the trusted compiler?

4.1 Terminology and notation

This paper focuses on compilers. For purposes of this paper, compilers execute in some environment, receiving as input *source code* as well as other input from the environment, and producing a result termed an *executable*. A compiler is, itself, an executable.

Figure 1 illustrates the notation used in this paper. A shaded box shows a compilation step, which executes a compiler (input from the top), processing source code (input from

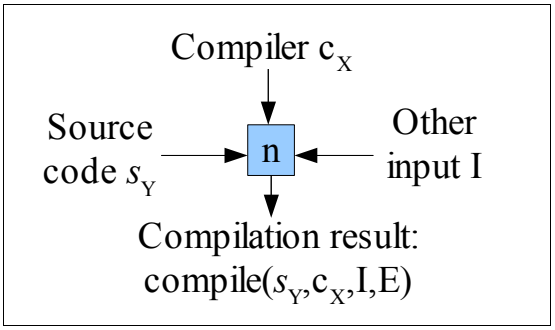

Figure 1: Illustration of graphical notation

the left), and uses other input (input from the right), all to produce a executable (output exiting down). To distinguish the different steps, each compilation step will be given a unique name (shown here as "n"). Source code that is purported to be the source code for the executable Y is notated as s_Y. The result of a compilation step using compiler X, source code s_Y, other input I, and in environment E is an executable, notated here as compile(s_Y, c_X, I, E). Where the environment can be determined from context (e.g., it is all the same) that parameter is omitted; where that is true and any other input (if relevant) can be inferred, both are omitted yielding the notation compile(s_Y, c_X) or just c(s_Y, c_X).

The widely-used "T-diagram" (aka "Bratman") notation is not used in this paper. T-diagrams were originally created by Bratman [Bratman1961], and later greatly extended and formalized by Earley and Sturgis [Earley1970]. T-diagrams can be very helpful when discussing certain kinds of bootstrapping approaches. However, they are not a universally perfect notation, and this paper intentionally uses a different notation because the weaknesses of T-diagrams make DDC unnecessarily difficult to describe:

1. T-diagrams combining multiple compilation steps can be very confusing [Mogensen2007, 219]. This is a serious problem when representing DDC, since DDC is fundamentally about multiple compilation steps.
2. T-diagrams quickly grow in width when multiple steps are involved; since paper is usually taller than it is wide, this can make complex situations more difficult to represent on the printed page. Again, applying DDC involves multiple steps.
3. T-diagrams do not handle multiple subcomponents well (e.g., a library embedded in a compiler). The notation can be "fudged" to do this (see [Early1970, 609]) but the resulting graphic is excessively complex. Again, compilation of real compilers using DDC often involves handling multiple subcomponents, making this weakness more important.
4. T-diagrams create unnecessary clutter when applied to DDC. In a T-diagram, every compiler source code and compiler executable, as well as their executions, are represented by a T. This creates unnecessary visual clutter, making it difficult to see what is executed and what is not.

Niklaus Wirth abandoned T-diagrams in his 1996 book on compilers, without even mentioning them [Wirth1996], so clearly T-diagrams are not absolutely required when discussing compiler bootstrapping. The notation of this paper uses a single, simple box for each execution of a compiler, instead of a trio of T-shaped figures. As DDC application becomes complex, this simplification matters.

4.2 Informal description of DDC

In brief, to perform DDC, source code must be compiled twice. First, use a separate "trusted" compiler to compile the source code of the "parent" of the compiler under test. Then, run that resulting executable to compile the purported source code of the compiler under test. Then, check if the final result is *exactly* identical to the original compiler executable (e.g., bit-for-bit equality) using some trusted means. If it is, then the purported source code and executable of the compiler under test correspond, given some assumptions to be discussed later.

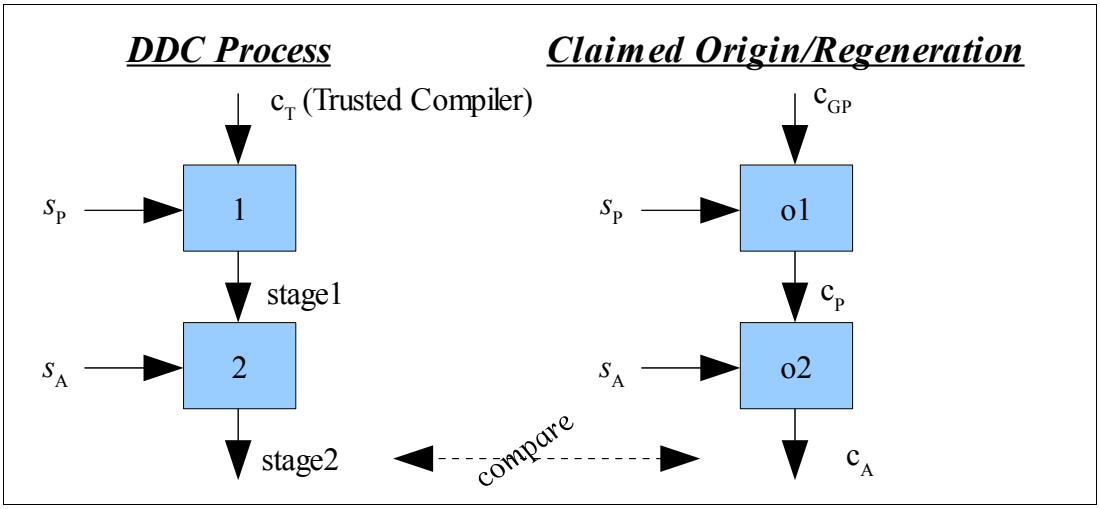

Figure 2: Informal graphical representation of DDC

Figure 2 presents an informal, simplified graphical representation of DDC, along with the claimed origin of the compiler under test (this claimed original process can be re-executed as a check for self-regeneration). The dashed line labeled "compare" is a comparison for exact equality. This figure uses the following symbols:

- c_A: Executable of the compiler under test, which may be corrupt (malicious compilers are by definition corrupt).
- s_A: Purported source code of compiler c_A. Our goal is determine if c_A and s_A correspond.
- c_P: Executable of the compiler that is purported to have generated c_A (it is the purported "parent" of c_A).
- s_P: Purported source code of parent c_P. Often a variant/older version of s_A.
- c_T: Executable of a "trusted" compiler, which must be able to compile s_P.. The exact meaning of "trusted" will be explained later.
- 1, 2, o1, o2: Stage identifiers. Each stage executes a compiler.
- stage1, stage2: The outputs of the DDC stages. Stage1 is a function of c_T and s_P, and can be represented as $c(s_P, c_T)$ where "c" means "compile". Similarly, stage2 can be represented as $c(s_A, \text{stage1})$ or $c(s_A, c(s_P, c_T))$.

The right-hand-side shows the process that purportedly generated the executable under test (c_A) in the first place. Since the graphical depiction of the process is identical, it should not be surprising that the results should be identical; one of the challenges resolved by this dissertation is to formally prove this (given certain conditions).

Before performing DDC itself, it is wise to perform a regeneration check, which checks to see if we can regenerate c_A using exactly the same process that was supposedly used to create it originally. Since c_A was supposed to have been created this way in the first place, regeneration should produce the same result. In practice, the author has found that

this is often not the case. For example, many organizations' configuration control systems do not record all the information necessary to accurately regenerate a compiled executable, and the ability to perform regeneration is necessary for the DDC process. In such cases, regeneration acts like the control of an experiment; it detects when we do not have proper control over all the relevant inputs or environment[5]. Malicious compilers can also pass the regeneration test, so by itself the regeneration test is not sufficient to reliably detect malicious compilers.

We then perform DDC by compiling twice. These two compilation steps are the origin of this technique's name: we compile twice, the first time using a different (diverse) trusted compiler. All compilation stages (stage 1 and stage 2, as well as the regeneration test) could be performed on the same or on different environments. Libraries can be handled in DDC by considering them as part of the compiler (if they are executed in that stage) or part of the source code (if they are used as input data but not executed in that stage).

Note that the DDC technique uses a separate trusted compiler as a check on the compiler under test. The trusting trust attack assumes that all later generations of the compiler will be descendants of a corrupted compiler; using a completely different second compiler can invalidate this assumption. The trusted compiler and its environment may be malicious, as long as that does not impact their result during DDC, and they may be very slow.

[5]DDC will not create an identical executable unless the regeneration check would succeed, and so from that perspective the regeneration check is mandatory. *Performing* the regeneration check has not been made mandatory, because there may be other evidence that clearly shows that the regeneration check would succeed. In most cases, however, using the regeneration check is *strongly* encouraged.

The formalized DDC model, along with formalized assumptions and its proof, are presented in chapter 5.

4.3 Informal assumptions

All approaches have assumptions. These will be formally and completely stated later, but a brief statement of some assumptions should help in understanding the approach:

1. We must have a trusted compiler c_T, comparer, and environment(s) used in DDC, and a trusted way to acquire c_A and s_A. In this paper, something is "trusted" if we have justified confidence that it does not have triggers and payloads that would affect the production of the compiler under test. They may have triggers and payloads, as long as they do not affect the result. It may have defects, though as shown later, any defects that affect its result in DDC are likely to be detected.
2. Compiler c_T must have the same semantics for the same constructs as required by s_P. Obviously, a Java(TM) compiler cannot be used directly as c_T if s_P is written in the C language! But if s_P uses any nonstandard language extensions, or depends on a construct not defined by a language specification, then c_T must implement them in the way required by s_P. Any defect in c_T can also cause problems if it affects compiling s_P (otherwise it is irrelevant for DDC).
3. The compiler defined by s_P should be deterministic given its inputs. That is, once compiled, and then executed multiple times given the same inputs, it should produce exactly the same outputs each time. If the compiler described by s_P is non-deterministic, in some cases it could be handled by running the process

multiple times, but it is often easier to control enough inputs to make the compiler deterministic. Note that the regeneration process is helpful in detecting undesired non-determinism.

The DDC technique does *not* assume that different compilers must produce the same executable output, given the same input. Indeed, compiler executables c_A, c_P, and c_T might run on or generate code for different CPU architectures.

4.4 Special case: Self-parenting compiler

An important special case is when $s_P=s_A$, that is, when the putative source code of the parent is the same as the putative source code of the compiler under test. There are often good reasons for releasing executables generated this way. For example, a compiler typically includes many optimization operations; each new version of a compiler may add new or improved optimization operations. By releasing a self-parented compiler, the supplier would release a compiler executable that uses the latest versions of those optimizations, giving the compiler itself maximum performance. Many existing compilers (including as GCC) use the compiler bootstrap test (essentially the self-regeneration check) to test themselves, so a compiler's build and test process may already include an automated way to create a self-parenting compiler. Figure 3 shows how figure 2 simplifies in this case.

Because this is a common case, the older paper [Wheeler2005] only considered this case. In contrast, this dissertation considers the more general case, subsuming self-parenting as a special case.

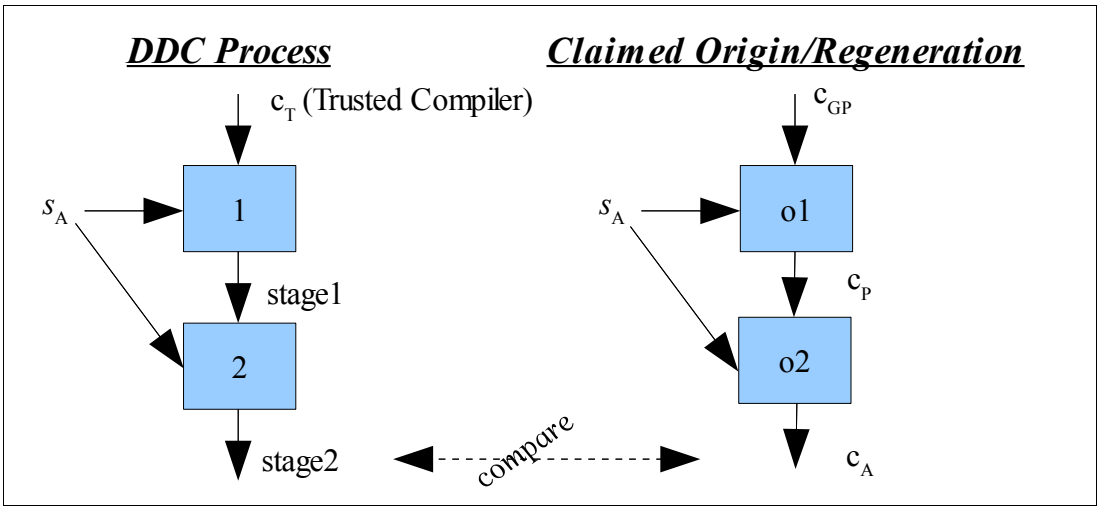

Figure 3: Informal graphical representation of DDC for self-regeneration case

Having a self-parenting compiler can simplify certain kinds of work. As discussed in more detail below, DDC can only show that source code and executable correspond, so review of compiler source code is still required. In the general case, the entire source code set (the union of s_A and s_P) must be reviewed. Sometimes the two sets of source code s_A and s_P are different but extremely similar, so reviewing only their differences (e.g., through tools such as "diff") is easy to do – but this is not true in general. Since s_A=s_P in a self-parented compiler, the union of s_A with s_P is the same as s_A alone, so reviewing the union of s_A and s_P is simpler—it only requires a review of s_A. Also, when a compiler is its own parent, a simplified regeneration check may be used to detect many problems without performing the complete regeneration test. This test, which can be termed "self-regeneration", is simply using c_A to compile its putative source code s_A; the regeneration is successful if the generated executable is the same as the original c_A.

It is still useful to be able to handle the general case. Compiler c_P need not be a radically different compiler; it might simply be an older version of c_A, differ only in its use of different compilation flags, or differ only in that it embeds a different version of a library executable. Nevertheless, if c_P and c_A are different, the general form of DDC must be used. Also, it is possible to have a "loop" of compilers that mutually depend on each other for self-regeneration. In this case, the more general form of DDC is needed to break the loop.

4.5 Why not always use the trusted compiler?

DDC uses a second "trusted" compiler c_T, which is "trusted" in the sense that it is very unlikely to have triggers or payloads that affect recompiling s_P and s_A. Given this informal explanation, we can now answer an obvious question: Why not always use the trusted compiler c_T?

In DDC, compiler c_T is only used to determine if executable c_A corresponds with its source code s_A. There are many reasons compiler c_T might not be suitable for general use. For example, compiler c_T may be slow, produce slow code, generate code for a different CPU architecture than desired, lack useful functions (the trusted compiler only needs to be able to compile source code s_P), be costly, or have undesirable software license restrictions. It is possible that the only purpose of the trusted compiler is to operate as a trusted checker for the more widely-used compiler, in fact, there are good reasons to do so. It is much easier to verify (and possibly formally prove) a simple compiler that has limited functionality and few optimizations; such compilers might not be suitable for

general production use, but would be ideal as trusted compilers used to check production compilers. The trusted compiler could even be a "secret" compiler that is never publicly released (as source, executable, or a service), specifically to make it difficult for an attacker to avoid detection by DDC.

Even if c_T is not suitable general use, it is true that c_T could be used as a "trusted bootstrap" compiler that would always be used as part of a bootstrap process to generate each new version of c_A. However, if we do this without using DDC, we have merely moved the trusting trust attack to a different location: We must now perfectly protect c_T and the bootstrap process used to create each new version of c_A. Should the protection of c_T ever fail, an attacker may change c_T into a malicious compiler, resulting in the corruption of future versions of c_A. By using DDC, we can use c_T as a separate check, requiring the attacker to subvert *two* separate compiler-creation processes to go undetected. Indeed, DDC could be performed multiple times using different compilers as c_T and/or different environments, requiring an attacker to subvert *all* of them (to go undetected) and thus greatly increasing confidence that c_A corresponds with s_A.

5 Formal proof

This section presents a formal proof of DDC. The first subsection presents a more complete graphical model of both the DDC process and how the compiler under test is claimed to have been created. This is followed by a description of the formal notation used, the tools used, the rationales used in proof steps, and other proof conventions. After this, the three key proofs are presented.

Each proof presents a set of predicates, functions, and assumptions about DDC in the formal notation, and shows how they lead to the concluding proof goal. The three proofs are named:

1. source_corresponds_to_executable: This is the key proof for DDC. It shows that given certain assumptions, if stage2 (the result of the DDC process) and c_A (the original compiler-under-test) are equal, then the executable c_A and the source code s_A exactly correspond.
2. always_equal: This proves that, under "normal conditions" (such as when compiler executables have not been rigged and thus *do* correspond to their respective source code), c_A and stage2 are in fact always equal. Thus, the first proof is actually useful, because its assumptions will often hold. This also implies that if c_A and stage2 are *not* equal, then at least one of its assumptions is *not* true.

3. cP_corresponds_to_sP: The previous "always_equal" proof does not require that a "grandparent" compiler exist, but having one is a common circumstance. This third proof shows that if there *is* a grandparent compiler, one of the assumptions of proof #2 can be proved given other assumptions that may be easier to verify (potentially making DDC even easier to apply in this common case).

5.1 Graphical model for formal proof

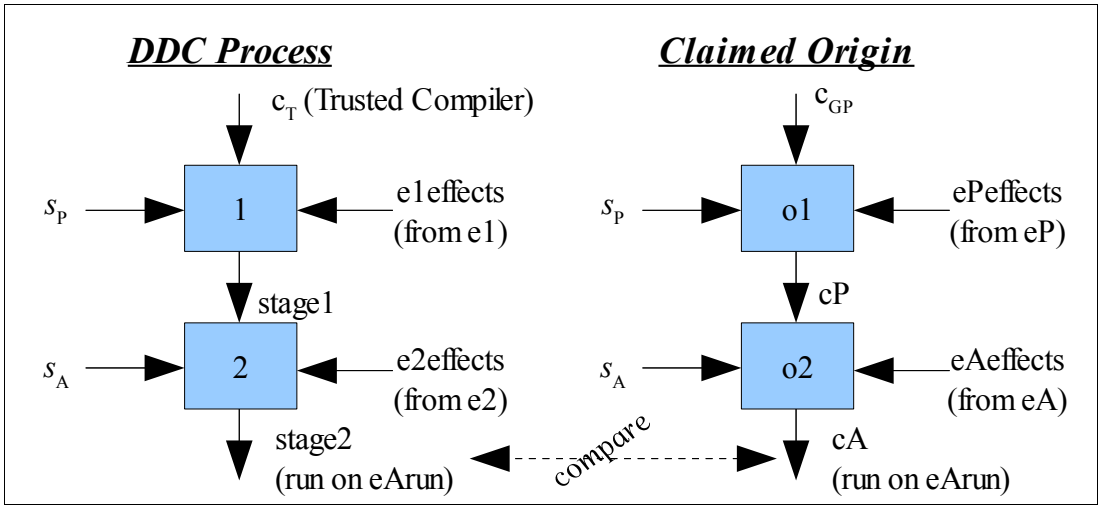

Figure 4: Graphical representation of DDC formal model

Figure 4 graphically represents the DDC stages and how the compiler under test c_A was putatively created. This is a more rigorous version of figure 2; the formal model includes more detail to accurately model potentially-different compilation environments and the effects these environments have on the compilation processes.

This dissertation argues that if the DDC process produces a "stage2" that is identical to the c_A, and certain other assumptions are true, then the executable stage2 corresponds to the source code s_A. The similarity of the DDC process and claimed origin figures suggest that this might be reasonable, but the challenge is to formalize exactly what those assumptions are, and then prove that this is true from those assumptions.

5.1.1 Types

Although types (sorts) are not directly used in the proof, it is easier to explain the graph and proofs by assigning types to the various constants used. There are four basic types:

1. *Data:* For our purposes, data is information that is used as source code (input) and/or is the resulting executable (output) of a compilation. Some of the data could be both source and executable (e.g., a library object file could be executed during compilation and also copied into the final executable). As implied by its definition, data can be:
    a. *Executable:* Data that can be executed by a computing environment. Compilers produce executables, and compilers themselves are executables.
    b. *Source*: Data that can be compiled by a compiler to produce an executable. Any source (aka source code) is written in some language.
2. *Environment:* A platform that can run executables. This would include the computer hardware (including the central processing unit) and any software that supports or could influence the compiler's result (e.g., the operating system). It could include a byte code interpreter or machine simulator.

3. *Language*: The language (used by some source) that defines the meaning of the source.
4. *Effects:* All information or execution timing arising from the environment that can affect the results of a compilation, but is not part of the input source code. This is used to model random number generators, thread execution ordering, differences between platforms allowed by the language, and so on. Note that this is not simply data in the usual sense, since other issues such as thread execution ordering are included as effects.

5.1.2 DDC components

The DDC process, as shown in figure 4, includes the following components, with the following types and meanings:

- c_T: Executable. The trusted compiler. It is trusted in the sense that it is trusted to not have triggers or payloads that will activate when compiling source s_P.
- s_P: Source. The (putative) source code of the "parent" compiler.
- s_A: Source. The (putative) source code of the compiler under test (c_A).
- e1: Environment. The environment that executes compilation step 1, which uses c_T to compile s_P and produce stage1.
- e2: Environment: The environment that executes compilation step 2, which uses stage1 to compile s_A and produce stage2.
- eArun: Environment: The environment that stage2 is intended to run on.
- e1effects: Effects. The effects sent from environment e1 to compilation step 1.

- e2effects: Effects. The effects sent from environment e2 to compilation step 2.
- stage1: Executable. The result of DDC compilation step 1. This will be defined, using the functional notation below, as compile(s_P, c_T, e1effects, e1, e2).
- stage2: Executable. The result of DDC compilation step 2. This will be defined as compile(s_A, stage1, e2effects, e2, eArun).

Note that s_A may be equal to s_P. Similarly, e1 may be equal to e2 or eArun, and e2 may be equal to eArun. These identities are permitted but not required by DDC. All processes (including the compilations and their underlying environments, the process for acquiring c_A, and the process for comparing c_A and stage2) must be trusted (i.e., they must not have triggers or payloads that affect their operation).

5.1.3 Claimed origin

The compiler under test c_A was putatively developed by a similar process. This "claimed origin" process can also be modeled, with the following components not already described in the DDC process:

- c_{GP}: Executable. The grandparent compiler, if there is one.
- eP: Environment. The environment that executes compilation step o1, which uses c_{GP} to compile source s_P and produce executable c_P.
- eA: Environment: The environment that executes compilation step o2, which uses c_P to compile s_A and produce c_A.
- ePeffects: Effects. The effects sent from e_P to compilation step o1.
- eAeffects: Effects. The effects sent from e_A to compilation step o2.

- c_P: Executable. Parent compiler.
- c_A: Executable. The compiler under test, which putatively was developed by the process above.

Note that compiler-under-test c_A may, in fact, be different than if it were really generated through this process. But if c_A *was* generated through this process, we can prove that certain outcomes will result, given certain assumptions, as described below.

5.2 Formal notation: First-Order Predicate Logic with Equality (FOPLE)

The logic notation used in this paper to develop the proofs is traditional first-order predicate logic with equality (FOPLE); it was selected because it is a widely understood and accepted logic system. In this paper, the notation and conventions defined in [Huth2004, 93-139] are used. FOPLE has two types of expression: a *term* and a *formula*.

A *term* (which denotes an object) is defined as: a variable, a constant, or a function application of form $f(\tau_1, \tau_2, ..., \tau_n)$ where each of the zero or more comma-separated parameters is a term. In this paper, variables begin with an uppercase letter, while constants begin with a lowercase letter (this is the same convention used by Prolog).

A *formula* (which denotes a truth value) is defined as: $\neg\Phi$, $\Phi \wedge \Psi$, $\Phi \vee \Psi$, $\Phi \rightarrow \Psi$, $\forall \chi \Phi$, $\tau_1 = \tau_2$, $\tau_1 \neq \tau_2$, or a predicate of form $p(\tau_1, \tau_2, ..., \tau_n)$ where each of the one or more comma-separated parameters is a term. This definition requires that Φ and Ψ are formulas, χ is an unbound variable, and anything beginning with τ is a term.

In some sense, a formula is a boolean that represents true or false, while a term represents any non-boolean type[6]. Functions and predicates have the same syntax if they have any parameters. Table 1 shows the traditional FOPLE notation for FOPLE expressions (terms and formulas), an equivalent ASCII representation, and a summary of its meaning:

Table 1: FOPLE notation

Traditional Notation	**ASCII Represent-ation**	**Meaning**
$\neg\Phi$	- PHI	*not* Φ, aka negation. If Φ is true, $\neg\Phi$ is false; if Φ is false, $\neg\Phi$ is true. $\neg\neg\Phi$ is equivalent to Φ.
$\Phi \wedge \Psi$	PHI & PSI	Φ *and* Ψ, aka conjunction and "logical and". Both Φ and Ψ must be true for the expression to be true.
$\Phi \vee \Psi$	PHI | PSI	Φ *or* Ψ, aka disjunction and "logical inclusive or". Φ, Ψ, or both must be true for the expression to be true.
$\Phi \rightarrow \Psi$	PHI -> PSI	Φ *implies* Ψ, aka implication, entailment, or "if Φ, then Ψ". Equivalent to $(\neg\Phi)\vee\Psi$.
$\forall \chi\, \Phi$	all Chi PHI	*For-all*, aka universal quantification. For all values of variable χ, Φ is true. In this paper, this is optional; all unbound variables are universally quantified.
$\tau_1 = \tau_2$	tau_1 = tau_2	τ_1 *equals* τ_2. If true, τ_2 can substitute for τ_1.
$\tau_1 \neq \tau_2$	tau_1 != tau_2	τ_1 *is not equal to* τ_2. Equivalent to $\neg(\Phi = \Psi)$.
$x(\tau_1, \tau_2, \ldots, \tau_n)$	x(tau_1, tau_2, ..., tau_n)	*Function or predicate x* with terms $\tau_1, \tau_2, \ldots, \tau_n$. A predicate is like a function that returns a boolean.

Parentheses are used to indicate precedence. FOPLE also has a "there exists" notation (using $\exists$) which is not directly used in this paper. In this paper, a top-level FOPLE formula is terminated by a terminating period (".").

[6]This exposes a weakness in traditional FOPLE as a notation—predicates and functions cannot have formulas (booleans) as parameters. For example, traditional FOPLE forbids the definition of a function *if_then_else(formula1, term1, term2)* that returns term1 if formula1 is true, else it returns term2. FOPLE also does not include built-in support for types (sorts), though there are extensions which do so. Nevertheless, these FOPLE weaknesses do not cause problems for the proof of DDC, and since FOPLE is widely-understood and widely-implemented, FOPLE is used.

For example, the following FOPLE formula could represent “all humans are mortal”:

```
human(X) -> mortal(X).
```

This formula can be read as “for all values of X, if X is human, then X is mortal”. Note that “X” is a variable, not a constant, because it begins with a capital letter. Also note that since X is not bound, an implied “all X ...” surrounds the entire formula.

In addition, the following formula could be used to represent “Socrates is human”:

```
human(socrates).
```

From these two formulas, it can be determined that “Socrates is mortal”:

```
mortal(socrates).
```

FOPLE is a widely-used general notation, and not designed for proofs about specific fields (such as compilation). Thus, as with most uses of FOPLE, additional “non-logical” symbols must be added before particular problems can be analyzed. In this paper, these additions are the various constant terms in the graphical model described in 5.1 (above), as well as various predicates and functions that will be defined below. The proofs below will introduce these predicates and functions, as well as various assumptions, and then show that certain important conclusions (termed “goals”) can be formally proved from them. Some assumptions define a term, predicate, or function; these assumptions are also called “definitions” to clarify what kind of assumption they are.

All formal models, including the one in this dissertation, must include lowest-level items (such as predicates, functions, and constants) that are not defined in the formal model itself. Therefore, it is unreasonable to protest that these lowest-level items are not defined in this model, since that is necessarily true. The key is that the lowest-level items

should accurately model the real world, thus forming a rational basis for proving something about the real world.

5.3 Tools: Prover9 and Ivy

Early versions of these proofs (which did not account for the differing environments) were developed by hand. Unfortunately, it was very difficult to rigorously check or amend those hand-created proofs[7]. The PVS Specification and Verification System was then used for some time, in part because it has a powerful notation that supports type-checking (which can eliminate some errors) and higher-order logic. At the time, it was thought that higher-order logic would be especially helpful, since a compiler can be viewed as a computational function that produces a computational function. However, while PVS is very good at what it does, and several proofs were created using PVS, PVS required a large amount of manual effort to produce the proofs. These early proofs showed that higher-order logic was not necessary or especially helpful in modeling this particular problem, and that other logic systems and provers could be easily used instead. These other tools had less powerful notations (e.g., just first-order logic without typing), but these tools could better automate proof development.

The final proofs, as presented in this dissertation, were developed and checked with the assistance of two tools, prover9 and ivy:

[7] The original hand-created proofs did not account for the possibility of different environments. When attempting to modify the proofs to account for the different environments, the painful "bookkeeping" required to keep the proof accurate soon led the author to look for an automated tool.

- Prover9 is an automated theorem prover for first-order and equational (classical) logic, which uses an ASCII representation of FOPLE. All of the proofs given in this section were developed by prover9 version Aug-2007.
- Ivy is a separate proof checker that can accept and verify the proof as output by prover9. Ivy is written in ACL2 and has itself been proven sound using ACL2. All of the prover9 proofs were verified by ivy. Indeed, one reason prover9 was chosen over some other tools was the availability of ivy.

There are many reasons to have very high confidence in these proofs. The proofs were verified by a separate tool (ivy) that itself has been proven sound. The source code for prover9, ivy, and ACL2 are all publicly visible under the terms of the GNU General Public License (GPL); this public visibility enables widespread public review. The proofs were hand-verified by the author, and then presented and reviewed by a team of people at the Institute for Defense Analyses (IDA), again increasing the likelihood that the proofs are correct.

Far more detail about prover9 is provided in [McCune2008]; its general approach (in particular, information on resolution and paramodulation) is discussed in detail in texts such as [Duffy1991]. For purposes of this paper, prover9 is given a set of assumptions and a goal statement, using first-order predicate logic with equality (FOPLE). Prover9 negates the goal, transforms all assumptions and the goal into simpler clauses, and then attempts to find a proof by contradiction. Should prover9's search algorithm find a proof, it can print the sequence of steps and the rationale for each step that leads to the proof.

Traditional FOPLE and the prover9 tool (which implements FOPLE) do not directly support types. It is possible to implement types (sorts) using FOPLE: types of constants can be declared as assertions (e.g., "executable(cA)" could represent "c_A is an executable"), assertions about compilers could be modified to state the types of compiler inputs and outputs, and the goal could be extended to include type requirements. However, because prover9 does not directly support type declaration, implementing types in prover9 makes the proofs far more complicated. These complications do not add value, because the types of compiler input and output are not in doubt (and thus do not need proof). As a result, types are only used in this paper as a way to clarify the proof results.

It should be noted that these tools did not make creating the proofs trivial. In particular, prover9 can only find a proof given a correct goal and assumptions. When prover9 cannot prove a goal, it eventually times out, and it is often difficult to determine *why* the proof cannot be found. Its companion tool Mace4 may be able to find a counter-example, but even then it is often not obvious what is wrong. Prover9 will also sometimes use information it does not need, leading to over-complicated proofs. To counteract this, each proof was developed separately and includes only the statements necessary for the proof.

5.4 Proof step rationales

Every step in the proofs has a rationale, which is one of the following (for clarity, the terminating "." in top-level formulas is omitted in this bulleted list):

- Assumption: Given assumption. All definitions are assumptions.
- Goal: The given goal to be proved.

- Clausify: Transform a previous step (formula) into a normalized clausal form. In particular, all expressions of the form $A \rightarrow B$ are transformed into $(\neg A) \vee B$. It also performs skolemization, transforming any "there exists" quantifier to function references, but this quantifier is not used in this paper. See [McCune2008] and [Duffy1991] for details.
- Copy...flip: Copy a previous result but reverse the order of an equality statement. Thus, given A=B, this rationale produces B=A.
- Deny: Negate a previous step; this processes the goal statement.
- Resolve: Resolution (aka general resolution), that is, produce a resolvant from two clauses. Resolution is a generalized version of ground (propositional) resolution, so to explain resolution, we will first explain ground resolution.

  Ground resolution is a derivation rule that applies to clauses in propositional logic (a simpler logic than FOPLE that lacks terms, predicates, functions, quantification (for-all and there-exists), and equality; variables are true or false). Ground resolution requires two ground clauses (formulas) of the form $C_1 \vee L \vee C_2$ and $D_1 \vee L' \vee D_2$, where L′ is a complement (negation) of L and at least one of C_1, C_2, D_1, and D_2 are non-empty. From that, ground resolution can derive $C_1 \vee C_2 \vee D_1 \vee D_2$ removing any duplicates (this can be informally viewed as combining the two clauses with L and L′ "canceling" each other out). For example, given both $P \vee Q$ and $\neg P \vee R$, resolution can derive $Q \vee R$. Ground resolution is a sound rule for reasoning because any L must be either true or false: If L is false, and $C_1 \vee L \vee C_2$ is true, then $C_1 \vee C_2$ must be true. Conversely, if L

is true, then L′ is false, and since $D_1 \vee L' \vee D_2$ is true, then $D_1 \vee D_2$ must be true. Since *either* $C_1 \vee C_2$ or $D_1 \vee D_2$ *must* be true, it follows that $C_1 \vee C_2 \vee D_1 \vee D_2$ is *always* true. The traditional logic rule *modus ponens (*given P and $P \rightarrow Q$, then Q) is a special case of ground resolution; $P \rightarrow Q$ can be rewritten (using clausify) as $\neg P \vee Q$, and ground resolution can combine $\neg P \vee Q$ with P to derive Q.

The full resolution rule extends ground resolution so that it can handle quantifiers and predicates. It does this by using unification, the process of replacing the variables in the expressions by terms to make the modified expressions identical to each other. In particular, the full resolution rule can replace variables with constants if the substituting constants can meet all the conditions of a clause. For details, see section 3.3 of [Duffy1991].

- Para: Paramodulation. The paramodulation rule adds support for the equality relation. It performs the replacement of an expression with another expression it is equal to, including any parameter substitutions. For example, given $f(d, e, X)$ and $f(A, B, C) = g(C, B, A)$, paramodulation can derive $g(X, e, d)$. The precise definition of this rule is complex, for example, it can handle cases where the equality must only be true under certain conditions. For details, see section 3.3.7 of [Duffy1991].

5.5 Proof conventions

The notation of prover9 only supports simple ASCII text, and does not directly support the Unicode characters for logic notation (such as →) nor subscripts (such as c_A). Thus, the ASCII representation is used for all prover9 representations and results below. Constants with subscripts are represented by simply appending the subscript value, e.g., c_A is notated as cA. Spaces and newlines are ignored, so they are inserted occasionally to improve readability. All successful prover9 proofs end with the conclusion "$F" (false). This means that prover9 was able to find a contradiction given the assumptions and the negation of the goal. Definitions are a kind of assumption; their names begin with "definition_" if they are of the form "constant = EXPRESSION", and begin with "define_" otherwise. In the prover9 proof, assumptions and goals are assigned names using the prover9 "label" attribute (not shown in this paper).

Each of the proofs below begins with a formal statement (using FOPLE formulas) of the goal to be proved, along with a textual explanation. This is followed by subsections that introduce the required predicates, functions, and assumptions, as well as restating the goal. The predicates and functions are described by showing the name and format of the predicate or function, along with its parameters (using initial capital letters), without an ending period ("."), followed by a textual explanation. The assumptions (including definitions) and goal are described using FOPLE formulas ending with a period, again followed by a textual explanation. These are followed by a prover9 proof (verified by ivy), which shows in a table format how the assumptions prove the goal (using proof by

contradiction). The table includes the rationale for each step. The prover9 proof is followed by additional discussion about that proof.

5.6 Proof #1: Goal source_corresponds_to_executable

The key proof for DDC is to show that, if stage2 (the result of the DDC process) and c_A (the original compiler-under-test) are equal, then the compiled executable c_A and the source code s_A exactly correspond. This goal is easily represented by the following formula (using ASCII representation) named source_corresponds_to_executable:

```
(stage2 = cA) -> exactly_correspond(cA, sA, lsA, eArun).
```

As with all formal proofs in this paper, this proof introduces various predicates, functions, and assumptions. Since this first proof is central to the entire paper, as each assumption is introduced it will be shown how it builds toward the final goal. This is followed by a prover9 table (showing how the assumptions prove the final goal) and a brief discussion.

5.6.1 Predicate "=" given two executables

The predicate "=" (equal-to, aka equality) is part of the goal statement; it compares two executables to determine if they are equal. It is an infix predicate with this form:

```
Executable1 = Executable2
```

For purposes of DDC, two executables are equal if they have *exactly* the same structure and values that are used by the environment when it runs either executable. When performing DDC, this test for equality must occur in an environment that is trusted to accurately report on the equality of two executables (i.e., the environment and program

implementing this equality test must not have triggers/payloads for the values tested), and the two executables being compared must have been acquired in a trustworthy way.

In a traditional operating system with a filesystem, an executable would normally be one or more files, where each file would be a stream of zero or more bytes as well as metadata controlling its execution (including the set of attributes determining if and how to run the file). The sequence of bytes must be identical (the same length and at each position the same value), and the metadata effecting execution must have the same effect in execution when transferred to its execution environment (e.g., the "execution" bit or equivalent must have the same value). The "have the same effect" phrase is stated here because differences that are *not* used by the environment during execution are irrelevant. In particular, many operating systems record "date written" as part of the metadata, and this would typically not be the same between different compilation runs. Nevertheless, as long as those differences do not effect program execution, they do not matter. Indeed, if the differences are only compared in certain ways, and those relationships are maintained, then they do not matter. Thus, if a "makefile" compares dates, but only to determine which files came before or later, the specific dates do not matter as long as the relationships are maintained. In practice, it is relatively easy to determine what metadata has an effect by examining the source code s_A and s_P; if the source code does not use it (directly or via calls to the environment), then given the other assumptions, the resulting stage2 executable from DDC will not invoke them either. This is because the DDC process (though not the original generation process) is required to not include triggers or payloads that affect the execution process (as discussed in section 3.2).

If the executables are S-expressions[8], the usual definition of S-expression equality is used: Atoms are only equal to themselves (so 5=5), NIL is only equal to itself, and lists are equal iff they have the same length and each of their elements are equal. NIL and an empty list are distinct if and only if the execution environment can distinguish them. We presume S-expressions are written out as text and read back before use (otherwise there are complications due to pointer equivalence).

Note that executable equality is a *stricter* relationship than executable *equivalence*. Two executables may be considered *equivalent* in an environment if they always produce equal outputs given equal inputs, even if their internal structure and/or values are different. Two executables that are equals are also equivalent, but the inverse is not necessarily true. Though not shown in this paper, this first proof (#1) could still be proven if the executables were merely equivalent instead of equal (indeed, compiler executables would only need to be equivalent for the source code input that they compile, though formally expressing this is complicated and confusing). Unfortunately, determining if two executables E1 and E2 are equivalent is undecidable in the general case. This is because if there was any decision procedure D capable of determining equivalence, it could be invoked by E1 and E2. If found equivalent they could perform different operations, and if found different they could act the same [Cohen1984, part 4]. Even in very special cases it is often difficult to determine the equivalence of two

[8]"S-expression" is short for "symbolic expression". It is a convention for representing semi-structured data in human-readable textual form, and is used for both code and data in Lisp. For our purposes, an S-expression may be an atom (a number, symbol, or special term NIL) or a list; a list contains 0 or more ordered S-expressions. The actual definition is more complex (involving CONS pairs), but this is not important for purposes of this paper.

executables. Instead of focusing on the difficult-to-determine equivalence relationship, we will instead focus on the stricter equality relationship, which is a far easier and more practical test to perform. Proof #2 and proof #3 will show that under certain common conditions, two executables will be equal (not just equivalent), so limiting proof #1 to equality does not significantly reduce its utility.

5.6.2 Predicate exactly_correspond

The goal statement makes no sense unless the predicate "exactly_correspond" is defined. Predicate "exactly_correspond" has the following parameters:

```
exactly_correspond(Executable, Source, Lang, RunOn)
```

This predicate is defined to be true if, and only if, the Executable *exactly* implements source code Source when (1) that Source is interpreted as language Lang and (2) the Executable is run on environment RunOn. For this predicate to be true, the Executable must not do anything more, anything less, or anything different than what is specified by Source (when interpreted as language Lang). Note that this does *not* require that Source is a perfect implementation of some abstractly-defined language. In section 5.6.8 we will define a condition that will make the predicate exactly_correspond true.

5.6.3 Predicate accurately_translates

A related predicate that must be defined is accurately_translates, with these parameters:

```
accurately_translates(Compiler, Lang, Source, EnvEffects, RunOn, Target)
```

This predicate is true if and only if the Compiler (an executable) correctly implements language Lang when compiling a particular Source and given input EnvEffects (from the

environment), when it is run on environment RunOn and targeting environment Target. The Target is the environment that the compiler generates code for (which need not be the same as the environment the compiler runs in). The EnvEffects parameter models variations in timing and inputs from the environment, and will be explained further in the definition of the "compile" function in section 5.6.5.

5.6.4 Assumption cT_compiles_sP

We must assume that the trusted compiler c_T is a compiler for language specification lsP (lsP specifies the language used by source code s_P), that c_T will accurately translate s_P when run in environment e1, and that c_T targets (generates code for) environment e2. This assumption is named cT_compiles_sP:

```
all EnvEffects accurately_translates(cT, lsP, sP, EnvEffects, e1, e2).
```

In short, c_T has to accurately implement the language lsP, at least sufficiently well to compile s_P. Otherwise, c_T can't be used to compile s_P. For example, if s_P was written in C++, then a Java compiler cannot be directly used as the trusted compiler c_T. Compiler c_T must not have triggers or payloads that activate when compiling s_P. Neither e1=e2 nor e1≠e2 is asserted; thus, e1 may but need not be the same as e2. The "all" in the formal statement is optional, but is included here for emphasis.

5.6.4.1 Implications for the language specification

This proof could have been created without mentioning languages at all; the formal model could simply require that (1) c_T will accurately translate s_P when run in environment e1 and that (2) c_T targets (generates code for) environment e2. However, it

would have been easy to misunderstand the proof results. For example, without noting the different languages, the proof could be easily misunderstood as requiring that all compilers implement the same language. Noting the languages clarifies that they *can* be different, and clarifies that the languages should be considered when performing DDC. Including the language in the proofs also provides a check on the proof that is similar to type-checking: The proof requires that in each compilation, the compiler used must support the language of the source code used as input.

The language specification lsP *must* include *all* of the syntactic and semantic requirements necessary to correctly interpret s_P. It *may*, but need not, include additional requirements not required to interpret s_P (as long as they do not interfere with interpreting s_P). In particular, lsP need not be the same as an official (standardized) specification, even if one exists. For example:

1. lsP may omit any requirements in an official specification, as long as the source code does not require them. So an official specification may include support for threading or floating point numbers, but if they are not needed when compiling the source code, then they can be safely omitted from lsP.
2. lsP may impose additional requirements that are explicitly left undefined in an official specification. For example, if the official language specification permits certain operations to be done in an arbitrary order (such as right-to-left or left-to-right evaluation of function parameters), but the given source code requires a particular order of evaluation to be correctly interpreted, then lsP must add the additional ordering requirement. Such additional requirements, if any, should be

included in the source code's documentation. It is usually *better* if the source code only requires what an official language specification guarantees, because there are likely to be more alternative compilers. But it's quite common for compiler sources to make assumptions that are not guaranteed by official specifications, and DDC can still be used in such cases.

3. lsP may impose additional length or size requirements than those imposed by an official specification. For example, if the source code requires support for identifier lengths, depth of parentheses, or size of result, then lsP includes those requirements.
4. If lsP includes ambiguous requirements, or requirements that are not fully defined, then those ambiguities or inadequate definitions must not matter when compiling the source code.
5. lsP may add various extensions as requirements that are not part of the official specification. Unsurprisingly, if the source code requires extensions, then the compiler used to compile that source code must somehow support those extensions.
6. lsP could even directly contravene an official specification on certain issues; what matters is what is required to correctly compile the source code.

The language lsP need not be formally specified, nor must it exist as a single document. If expressed, it is likely to take the form of a reference to an existing language standard combined with a description of the permitted omissions, the changes, and the additions.

The "language" may even be a set of languages, including a language for selecting which other language to use (e.g., the file extension conventions used for selecting between languages). For example, the GNAT Ada compiler's front-end is written in Ada, while the rest of the compiler is written in C; a trusted compiler for GNAT would need to be able to compile both Ada and C.

5.6.4.2 Implications for the trusted compiler and its environment

Compiler c_T need not implement a whole language, as defined by an official language specification—it only needs to implement what is required to compile s_P. So c_T may be a very limited compiler. In some cases, some compiler c_Q may only be suitable for use as a part of trusted compiler c_T if the source code goes through a preprocessor, or if the resulting executable goes through a postprocessor. For example, a preprocessor may be needed to convert nonstandard constructs into constructs that c_Q can handle, or perhaps c_Q implements a different specification (e.g., it may implement "K&R C" instead of the newer "ANSI C" specification). In this case, the compiler c_T is actually the combination of the preprocessor and c_Q. In theory there's no limit to how many steps can be chained together to construct c_T, but since they are all part of the trusted compiler they must be sufficiently trustworthy to meet the assumptions of the proof. In practice, these steps (including pre- and post-processors) should be limited, to limit the tools that are granted such trust.

Note that the trusted compiler (c_T) and the environment it executes on (e1) do *not* need to be completely defect-free nor non-malicious. This is important, since defect-free

compilers and environments are rare, and ensuring absolute non-maliciousness is difficult. Compiler c_T or environment e1 may be full of bugs, and/or full of triggers and payloads for inserting malicious code into other programs (including itself). We merely require that c_T, when executed on e1, perform an accurate translation when it compiles exactly one program's source code: s_P. So c_T may have defects – but they must not affect compiling s_P. Similarly, c_T may have malicious triggers and payloads – but c_T must not have triggers for s_P, or if it does, its payloads must not affect the results. Various real-world actions, such spot-checking or formally verifying the compiler executable c_T, can increase confidence that this assumption is true in the real world. In some cases, a secret compiler (where reading/writing its source, reading/writing its executable, and using it as a service is expressly limited to very few trusted people) may be useful as the trusted compiler; via DDC, it can be used to greatly increase confidence in the publicly-available compiler.

There is a subtlety in the formal model that is normally handled correctly by compiler users, but is noted here for completeness. That subtlety is that when performing DDC, we typically need to have different build instructions (as executed by the "real" compilers and environment) than when s_P and s_A were originally compiled. At first glance this appears to be a problem, because in the formal model of DDC, the source code s_P and s_A that is used in DDC must be *exactly* the same as the source code used in its original purported creation process. Yet the source code may include build instructions, indeed, nontrivial compilers often include complex build instructions as part of their source code. But if the build instructions are part of the source code, and the build instructions invoke

a compiler other than c_T, how can trusted compiler c_T be invoked during DDC? Similarly, if the environments e1 or e2 are different than the environments eP and eA (respectively), and/or if the option flags are different between compilers, how are these changes modeled? And similarly, if the build systems are substantially different (e.g., there are different build languages), how can we accurately model translating the build language? One solution is to consider the build instructions as not included in the source code, but this is grossly unrealistic for larger compilers with complex build instructions.

A better alternative that completely models these circumstances is to consider the build instructions to be part of the source code, and instead consider the trusted compiler c_T to be some "real" compiler c_T' plus a preprocessor. This preprocessor is trusted to correctly change the build instructions in a way that meets this assumption, e.g., so that the compilation process invokes c_T' instead of the original compilation process. In practice, this preprocessor is likely to be implemented by a human who modifies the build process (e.g., by setting an environment variable or make invocation to set the compiler to be used, modifying a makefile, or hand-translating the build instructions to a different build language). This step is so "obvious" to most compiler users that it would not normally be remarked on. Often this transformation is so simple that it is easy to forget that it even occurred. Nevertheless, by acknowledging this step, the formal model of DDC can accurately model what actually occurs. Since it is part of the trusted compiler c_T, this preprocessor step must be trusted to not include triggers and payloads that would effect the DDC compilation.

In general, the internal structure of trusted compiler c_T is irrelevant for the proof. Many problems in applying DDC (including modeling necessary changes to the build process as noted above) can be resolved by combining various processes (including preprocessors and/or postprocessors) as necessary to produce the final trusted compiler c_T. The only requirement is that all required assumptions (including the definitions) are met.

5.6.5 Function compile

Unsurprisingly, we must model compiling a program. We will model compiling as a function that returns an executable (a kind of data) and has the following parameters:

```
compile(Source, Compiler, EnvEffects, RunOn, Target)
```

This represents compiling Source with the Compiler, running the compiler in environment RunOn, and instructing the compiler to generate an executable for the target environment Target. Note that Target may or may not be the same as RunOn.

The parameter "EnvEffects" overcomes an issue in typical mathematical notation. In typical mathematical notation, a function provided with the same inputs will always produce the same outputs. Without the "EnvEffects" parameter, this would imply that a given compiler executable, when given the same Source, RunOn, and Target, will always produce exactly the same output (i.e., that it is *deterministic*). Unfortunately, this is *not* always true for all compilers. Some compilers *will* produce different outputs at different times, even when given the same source code. The reason is that environments can provide "effects", which are essentially inputs to the compilation process that affect the

outcome but are not part of the source code. Examples of effects that can cause non-determinism are:

1. Random number generators. A compiler's code generator or optimizer might have multiple alternatives, and instead of picking one deterministically, it might call on a random number generator to make that determination. If the environment provides different random numbers each time it is run, the results might be different. Note that under certain circumstances the GCC compiler will use a random number generator, but GCC also allows users to select a seed; if a seed is selected, then the sequence is deterministic and not random at all.
2. Heap allocation address values. Many systems today randomize addresses (e.g., of the heap or stack), in an attempt to counter attackers by making certain kinds of attacks harder to perform. However, a compiler's output may be changed by different address values. For example, some Java compilers use heap allocation addresses for hash calculation, and then use those hash values to control the sort order of some output. As a result, the output ordering may be different between executions, even given the same source code, execution environment, and target environment.
3. Execution order due to threading. Some compilers are multi-threaded and are only loosely ordered. The environment may execute the threads in a different order in different executions, and depending on the compiler, this may affect the output.

Thus, EnvEffects models the inputs from the environment which may vary between executions while still conforming to the language definition as used by Source.

As noted earlier, libraries may be modeled by considering them as part of the compiler (if they are executed) or part of the source (if they are used as input data but not executed).

In some discussions of DDC, we will occasionally use the simpler definition:

```
compile(Source, Compiler)
```

Of course, this definition cannot represent the different environments (RunOn and Target), nor can it represent the possibility that some programs are nondeterministic (which is modeled by EnvEffects), but in some situations these can be inferred from context. In some cases the function name "c" is used as an abbreviation for "compile".

5.6.6 Assumption sP_compiles_sA

We must assume that the source code s_P (written in language lsP) defines a compiler that, if accurately compiled, would be suitable for compiling s_A. To formally state this, we will assert that if we have some GoodCompilerLangP with the right properties, then using GoodCompilerLangP on s_P will produce a suitable executable:

```
accurately_translates(  GoodCompilerLangP, lsP, sP,
                        EnvEffectsMakeP, ExecEnv, TargetEnv) ->
   accurately_translates(
      compile(   sP, GoodCompilerLangP, EnvEffectsMakeP,
                 ExecEnv, TargetEnv),
      lsA, sA, EnvEffectsP, TargetEnv, eArun).
```

Strictly speaking, the name "sP_compiles_sA" is misleading; there is no guarantee that source code can be directly executed. However, more-accurate names[9] tend to be very long and thus hard to read.

Note that by combining this assumption (sP_compiles_sA) and the previous assumption cT_compiles_sP, we can determine a new derived result which we will name sP_compiles_sA_result:

```
accurately_translates(  compile(sP, cT, EnvEffectsMakeP, e1, e2),
                        lsA, sA, EnvEffectsP, e2, eArun).
```

Note that EnvEffectsMakeP and EnvEffectsP are not bound to any particular value, so they have an implicit "for all" around them. Since their actual values do not matter, to simplify these expressions they (and similar dummy values) can be replaced with arbitrary capital letters:

```
accurately_translates(compile(sP, cT, A, e1, e2), lsA, sA, B, e2, eArun).
```

Note that s_P (when compiled) does not need to implement the *whole* language s_A was written in, as defined by some official language standard. Instead, a compiled form of s_P only needs to implement the syntax and semantics of the language that s_A requires. The language specification lsA *must* include *all* of the syntactic and semantic requirements necessary to correctly interpret s_A; it *may*, but need not, include additional requirements not required to interpret s_A. This is fundamentally the same kind of relationship between c_T and s_P through language lsP as described in section 5.6.4, and the same explanation regarding language applies.

[9] Such as "sP_when_accurately_compiled_compiles_sA"

5.6.7 Definition definition_stage1

We must now begin to define the DDC process itself in this formal notation. As shown in figure 4, the executable "stage1" is created by compiling s_P using c_T, running on environment e1 and targeting environment e2. We will name this definition_stage1, and it is formally notated as:

```
stage1 = compile(sP, cT, e1effects, e1, e2).
```

Combining this with sP_compiles_sA_result, we find this result which we will name as definition_stage1_result1:

```
accurately_translates(stage1, lsA, sA, A, e2, eArun).
```

5.6.8 Definition define_exactly_correspond

There is a key relationship between the predicates "exactly_correspond" and "accurately_translates" that has not yet been expressed, which also provides insight into what it means when a source and executable exactly correspond. Fundamentally, if some Source (written in language Lang) is compiled by a compiler that accurately translates it, then the resulting executable exactly corresponds to the original Source. This relationship is named define_exactly_correspond, and is so central to the notion of "exactly_correspond" that it essentially defines it. This is expressed as:

```
accurately_translates(Compiler, Lang, Source, EnvEffects, ExecEnv, TargetEnv)
->
exactly_correspond(compile(Source, Compiler, EnvEffects, ExecEnv, TargetEnv),
Source, Lang, TargetEnv).
```

Combining this with the previous result, we can now determine a result that we will name define_exactly_corresponds_result1:

```
exactly_correspond(compile(sA, stage1, A, e2, eArun), sA, lsA, eArun).
```

5.6.9 Definition definition_stage2

We now introduce a formal model for how the DDC process generates stage2, which compiles source s_A using the executable stage1 and targets environment eArun:

```
stage2 = compile(sA, stage1, e2effects, e2, eArun).
```

Using the previous result, we can now determine definition_stage2_result1:

```
exactly_correspond(stage2, sA, lsA, eArun).
```

5.6.10 Goal source_corresponds_to_executable

We can now prove our goal, source_corresponds_to_executable. Recall that this goal is:

```
(stage2 = cA) -> exactly_correspond(cA, sA, lsA, eArun).
```

But we already know, per definition_stage2_result1, that:

```
exactly_correspond(stage2, sA, lsA, eArun).
```

If stage2 is exactly the same as c_A (the left side of the goal's implication), then we can replace stage2 with c_A, producing:

```
exactly_correspond(cA, sA, lsA, eArun).
```

QED.

5.6.11 Prover9 proof of source_corresponds_to_executable

Table 2 presents the proof found by prover9:

Table 2: Proof #1 (source_corresponds_to_executable) in prover9 format

#	Formula	Rationale
1	accurately_translates(A,B,C,D,E,F) -> exactly_correspond(compile(C,A,D,E,F),C,B,F)	Assumption define_exactly_correspond
2	(all A accurately_translates(cT,lsP,sP,A,e1,e2))	Assumption cT_compiles_sP
3	accurately_translates(A,lsP,sP,B,C,D) -> accurately_translates(compile(sP,A,B,C,D),lsA,sA,E,D,eArun)	Assumption sP_compiles_sA
4	stage2 = cA -> exactly_correspond(cA,sA,lsA,eArun)	Goal source_corresponds_to_executable
5	-accurately_translates(A,B,C,D,E,F) | exactly_correspond(compile(C,A,D,E,F),C,B,F)	Clausify 1
6	accurately_translates(cT,lsP,sP,A,e1,e2)	Clausify 2
7	-accurately_translates(A,lsP,sP,B,C,D) | accurately_translates(compile(sP,A,B,C,D),lsA,sA,E,D,eArun)	Clausify 3
8	stage1 = compile(sP,cT,e1effects,e1,e2)	Assumption definition_stage1
9	compile(sP,cT,e1effects,e1,e2) = stage1	Copy 8, flip
10	stage2 = compile(sA,stage1,e2effects,e2,eArun)	Assumption definition_stage2
11	compile(sA,stage1,e2effects,e2,eArun) = stage2	Copy 10, flip
12	cA = stage2	Deny 4
13	-exactly_correspond(cA,sA,lsA,eArun)	Deny 4
14	-exactly_correspond(stage2,sA,lsA,eArun)	Para 12 13
15	accurately_translates(compile(sP,cT,A,e1,e2),lsA,sA,B,e2,eArun)	Resolve 7 6
16	accurately_translates(stage1,lsA,sA,A,e2,eArun)	Para 9 15
17	exactly_correspond(compile(sA,stage1,A,e2,eArun),sA,lsA,eArun)	Resolve 5 16
18	exactly_correspond(stage2,sA,lsA,eArun)	Para 11 17
19	$F	Resolve 18 14

5.6.12 Discussion of proof #1

The existence of stage1 and stage2 implies termination of the compilation processes that produced them. This doesn't limit the proof's utility in the real world; a compilation

process that never finished would not be considered useful, and would certainly be noticed. Termination implies that s_A and s_P are computable and implementable, which in turn implies that the subset of languages lsA and lsP correspondingly used by s_A and s_P are also computable and implementable. Thus, given the assumptions above, s_A cannot call impossible functions like "return_last_digit_of_pi()". The languages lsP and lsA may have many additional capabilities, but for DDC only the proof assumptions are required.

Reviewers often search to see if a proof works given "null" or "absurdly small" cases. Oddly enough, the proof is still correct in these cases. It is theoretically possible that one or more of the compilers could be a one-byte value, a one-bit value, or even null, if the underlying environment implemented those values according to the proof assumptions. For example, an environment could theoretically have a single instruction that meant "compile", or it might even implement a "compile" function if it receives an empty sequence. This is rather hypothetical; real environments are very unlikely to work this way. However, there's no need to *prevent* this possibility, so the proof permits it.

The goal statement compares for equality between stage2 and c_A. As noted above, this requires that equality be correctly implemented; if the equality-checking program is itself subverted, this proof would not apply, so the equality-checking program and the environment it runs on must not be subverted. Similarly, the values stage2 and c_A that are compared must be acquired in a trusted manner; if the programs or environment used to copy them are subverted, then again, the proof will not apply (because the values the proof applies to might not be what is being tested).

Note that the converse of the proof #1's goal does not necessarily hold. The converse is:

```
exactly_correspond(cA,sA,lsA,eArun) -> stage2 = cA
```

There are many reasons the converse need not be true. For example, executable c_A might have been modified by adding extra unused information at its end, or had "no-operation" statements inserted into it that do not change the outputs it produces. Indeed, c_A could have been produced by compiling s_A using a different but trustworthy compiler and environment. In all these cases, c_A could exactly correspond to s_A, even though stage2 is not equal to c_A. But there *is* a common circumstance where stage2 and c_A must be equal; showing this is true is the focus of proof #2.

5.7 Proof #2: Goal always_equal

The first proof (source_corresponds_to_executable) shows that if c_A and stage2 are equal, then c_A and s_A exactly correspond. However, this first proof is not practically useful if c_A and stage2 are not normally equal. So we will next prove that, under "normal conditions", c_A and stage2 are in fact always equal. "Normal conditions" is expressed more formally below, but in particular, this includes the presumption that the compiler executables have *not* been tampered with (i.e., that the compiler executables correspond to their source code). This proof goal is named "always_equal", and is simply:

```
cA = stage2.
```

This second proof requires many more assumptions than the previous proof (10 instead of 5). It reuses 4 of the previous assumptions: definition_stage1, definition_stage2, cT_compiles_sP, and define_exactly_correspond. We do not need the assumption sP_compiles_sA for this proof; if s_P terminates but fails to compile s_A, the results will still

be equal (in this case the processes will produce equal error messages, which is probably not useful but it does not invalidate the proof). The new assumptions are definition_cA, cP_corresponds_to_sP, define_compile, sP_deterministic, sP_portable, and define_determinism, as defined below. These assumptions will carefully avoid using or making any assumptions about cGP, a possible "grandparent" compiler, since in some cases there may not *be* a grandparent compiler. Proof #3, to follow, will examine the common case when there *is* a grandparent compiler.

In this second proof, the predicates, functions, and assumptions will now be presented, along with their ramifications. This will be followed by the complete prover9 proof and a discussion.

5.7.1 Reused definitions define_exactly_correspond, definition_stage1, and definition_stage2

We will reuse several definitions. Here is definition define_exactly_correspond:

```
accurately_translates(Compiler, Lang, Source, EnvEffects, ExecEnv,
   TargetEnv) ->
exactly_correspond(compile(Source, Compiler, EnvEffects, ExecEnv,
TargetEnv), Source, Lang, TargetEnv).
```

Definition definition_stage1:

```
stage1 = compile(sP, cT, e1effects, e1, e2).
```

Definition definition_stage2:

```
stage2 = compile(sA, stage1, e2effects, e2, eArun).
```

5.7.2 Assumption cT_compiles_sP

We will also reuse assumption cT_compiles_sP:

```
all EnvEffects accurately_translates(cT, lsP, sP, EnvEffects, e1, e2).
```

5.7.3 Predicate deterministic

We will now define a new predicate:

```
deterministic(Source, Language)
```

This predicate is defined to be true if, and only if, the given source (when compiled by a compiler for Language) will be a deterministic executable. A deterministic executable always produces the same outputs, given the same inputs. This means that Source does not use any potentially non-deterministic capabilities in Language, such as a random number generator, requiring that memory addresses will be the same on all executions, vary its output depending on the scheduling of its threads, and so on.

A compiler need not be deterministic. For example, when there are optimization alternatives, it could "flip a coin" by calling on a random number generator in the environment, and produce different results each time. Or it could use hash tables based on object addresses, and then use those hash tables to determine the ordering of its output.

However, most compilers are deterministic, or can be executed in a way that makes them deterministic, because it is much more difficult to test non-deterministic compilers. Indeed, some compilers (such as GCC) use self-regeneration as a self-test—and such tests require determinism. For example, GCC's C++ compiler includes the ability to

control the random number seed used during compilation, specifically to cause its nondeterministic behavior to become deterministic. One exception is embedded timestamps: Some object code formats embed compilation timestamps in the file. However, such timestamps are common only for intermediate formats; one easy solution is to only compare final results, if they do not include embedded timestamps.

Note that this has nothing to do with non-determinism of the underlying CPU. The CPU can have all sorts of non-deterministic actions (e.g., using multiple cores). But if the CPU were so non-deterministic that it could not reliably write data in a particular order, even when programmed correctly, it is unlikely that a compiler (or any other program) would run on it.

5.7.4 Predicate portable

We will define a new predicate:

```
portable(Source, Language)
```

This predicate is defined to be true if, and only if, the given Source (when compiled by a compiler for Language) uses *only* constructs that are defined to be portable by all implementations of Language.

Many real-world languages include intentionally non-portable constructs that provide direct access to the underlying environment and/or use compiler extensions not supported by other compilers. For example, languages may provide nonstandard methods for opening files. However, we will need to compile the same program using different compilers, in potentially different environments. Thus, we must avoid such constructs, or

add those additional requirements to the language and ensure that all the implementations used in DDC and the origin of the compiler support them.

5.7.5 Function run

Previously we could treat compiling as a "black box", but for this proof more detail about compilation is needed. In particular, we must model executing a program. Thus:

```
run(Executable, Input, EnvEffects, Environment)
```

is a function that returns data. This data (the output) is the result of running Executable in Environment, giving it Input and the various environmental effects EnvEffects. The parameter "EnvEffects" models whatever the language allows the environment to vary that could have an effect on the results of running Executable, such as random number generator values or thread scheduling.

The results include standard out, standard error, and any files (file names, locations, and contents) generated or modified by its execution. Since different runs could have different environment effects as input (e.g., the random number generator from the environment might produce something different), it is possible that running the same executable with the same Input could produce different results.

5.7.6 Function converttext

Function converttext models an unfortunate complicating issue in the real world: Different systems encode text in different ways. Function

```
converttext(Data, Environment1, Environment2)
```

takes Data, where all text is in the standard format of Environment1, and returns the same Data but with all text converted to the standard format of Environment2.

In particular, a new line may be encoded differently by different environments. Common conventions, and users of those conventions, include:

- Linefeed (#x0A): Unix, GNU/Linux, MacOS X, Multics.
- Carriage Return (#x0D) : Apple II family, MacOS version 9 and before.
- Carriage return + Linefeed (#x0D #x0A): CP/M, MS-DOS, Microsoft Windows.
- NEL (#x85): OS/390 [Malaika2001].

Similarly, not all computer systems encode characters the same way; they may use ASCII, UTF-8, UTF-16 (which may be little-endian or big-endian), a locale-specific encoding (of which there are many), or even EBCDIC.

Since we will later compare values for exact equivalence, modeling these differences is necessary.

5.7.7 Function extract

Function extract accepts data, and returns a subset of that data:

```
extract(Data)
```

More specifically, function extract() extracts *only* the executable produced by a compiler, and silently throws away the rest (e.g., warning and error reports made during the

compilation process). A compilation process runs a compiler, and a compiler produces many outputs – but we only want the data that will be later used for execution. In a typical compilation environment, extract() will produce just the generated executable files, and not outputs to standard out, standard error, and/or log files.

5.7.8 Function retarget

Function retarget accepts source and target, and returns possibly modified source:

```
retarget(Source, Target)
```

Retarget represents any modifications to the source code Source that are necessary to change it so it will compile to run on the target environment Target. In many circumstances, Source will include various flags to the compiler that determine what environment the compiled executable will run on. If a different execution environment is to be used, the Source may need to be modified. If no such modifications are needed, retarget simply returns Source.

5.7.9 Assumption sP_deterministic

We will assume that source s_P, when compiled, describes a deterministic program:

```
deterministic(sP, lsP).
```

This means that source s_P either avoids all non-deterministic capabilities of language lsP, or uses them only in ways that will not affect the output of the program. For example, it can use threads, but if it does it will use mechanisms (such as locks) to ensure that race conditions will be avoided sufficiently to ensure that the output will be the same on each

execution given the same input. In some cases, setting the random number seed and algorithm for "randomness" may be necessary to ensure determinism.

Strictly speaking, this is an overly strong requirement; we only really *require* that s_P be deterministic when it's compiling s_A, and only in certain environments. But expressing that nuance makes the proof more complex, and doesn't add anything. Generally, compiler developers want their compilers to be deterministic, and not only when compiling specific programs or only when running on specific environments. So while this is a stronger requirement than is strictly necessary for the proof, it's a realistic assumption that real compilers tend to meet.

Note that we do *not* require that c_T or the grandparent compiler c_{GP} (if it exists) be deterministic. They *could* be deterministic, and often will be, but it is not necessary.

5.7.10 Assumption sP_portable

We will also assume that source s_P is portable:

```
portable(sP, lsP).
```

This means that source s_P avoids all non-portable capabilities of language lsP, or uses them only in ways that will not affect the output of the program. Again, this is a stronger assumption than strictly necessary, since this is really only necessary when it is used in the DDC process. However, compiler-writers often *do* try to limit the use of nonportable constructs in their compilers' source code, so this is a reasonable requirement.

5.7.11 Definition define_determinism

Simply stating that some source code describes a deterministic and portable program is not enough. What is the ramification of having source code that is deterministic and portable?

The answer is that, if the source code uses only the deterministic, portable capabilities of a language, and given two executables that exactly correspond to that same source code (possibly running in different environments), then those executables–when given the same input–will produce the same output. This is expressed as follows:

```
( deterministic(Source, Language) & portable(Source, Language) &
exactly_correspond(Executable1, Source, Language, Environment1) &
exactly_correspond(Executable2, Source, Language, Environment2)) ->
  ( converttext(run(Executable1, Input, EnvEffects1, Environment1),
                Environment1, Target) =
    converttext(run(Executable2, Input, EnvEffects2, Environment2),
                Environment2, Target)).
```

This is perhaps best explained by example. Imagine two properly-working C compilers, both of which are given this source code to print the result of calculating 2+2:

```
#include <stdio.h>
main() {
   printf("%d\n", 2+2);
}
```

The outputs of those two compilers is almost certain to be completely different, but *running* these two programs on their respective environments must produce the same result for this line, once their text output is converted into the same environmental format. Obviously, this depends on them implementing the same language (for the purposes of the given Source).

Combining this with sP_deterministic and sP_portable, we find the expression define_determinism_result:

```
-exactly_correspond(A,sP,lsP,B) | -exactly_correspond(C,sP,lsP,D) |
converttext(run(C,E,F,D),D,V) = converttext(run(A,E,W,B),B,V)
```

5.7.12 Assumption cP_corresponds_to_sP

How was compiler under test c_A created? The putative origin of c_A is that it was compiled by compiler c_P, and that c_P's executable exactly corresponds to source s_P. For the moment, we will simply assume this:

```
exactly_correspond(cP, sP, lsP, eA).
```

In many cases c_P will have been created by compiling s_P using some grandparent compiler c_{GP}. Proof #3 will show that this assumption (cP_corresponds_to_sP) can be proven given certain other plausible assumptions, including the existence of a grandparent compiler. However, by making this a simple assumption in proof #2, proof #2 is more general. For example, it is possible that c_P was created by hand-translating s_P into an executable; in this case, there may be no executable that is the grandparent compiler (since a human acted as the grandparent compiler), yet it may still be possible to accept this assumption.

5.7.13 Definition define_compile

In the previous proof we had simply accepted "compile" as a function that produced data:

```
compile(Source, Compiler, EnvEffects, RunOn, Target)
```

This represents compiling Source with the Compiler, running it in environment RunOn, but targeting the result for environment Target.

However, for this proof, more detail about the compilation process is needed, so the compilation process will now be modeled using more primitive functions:

```
compile(Source, Compiler, EnvEffects, RunOn, Target) =
   extract(converttext(run(Compiler, retarget(Source, Target),
           EnvEffects, RunOn), RunOn, Target)).
```

This is easier to explain by beginning on the right-hand-side, going from the inside expressions out. First, the Source is retargeted so that it will compile for environment Target (this includes changing compiler flags for the new target). Then run the Compiler on the environment RunOn with the retargeted Source code as input; note that if Compiler is a nondeterministic compiler, the environmental EnvEffects may have an effect on the results. The output will probably include text results (such as warnings, errors, and possibly the resulting executable depending on the kind of compiler it is). This text is then converted to Target's standard text format. Finally, the portions of the compilation results that can be run later are extracted; the rest of the material (such as warning text) is thrown away.

In practice, you only need to perform the converttext work on text that will be extracted; if it will be thrown away, then there's no need to actually perform the conversion. But this is merely an optimization, and is not necessary for the proof; it was easier to model in the way shown above.

5.7.14 Definition definition_cA

How was compiler under test c_A generated? Putatively it was generated by compiling source s_A, using compiler c_P. This is easily modeled, in a manner similar to stage1 and stage2:

```
cA = compile(sA, cP, eAeffects, eA, eArun).
```

It's quite possible that this assumption is not true, e.g., perhaps the executable of the compiler-under-test was recently replaced by a corrupt executable (such as a malicious executable). But for proof #2, we are considering what happens in the "benign" circumstance (where the putative origins are true), to show that a benign environment *must* produce a match.

5.7.15 Goal always_equal

Recall that the goal is to prove, given the preceding assumptions:

```
cA = stage2.
```

5.7.16 Prover9 proof of always_equal

Table 3 presents the proof found by prover9:

Table 3: Proof #2 (always_equal) in prover9 format

#	Formula	Rationale
1	deterministic(A,B) & portable(A,B) & exactly_correspond(C,A,B,D) & exactly_correspond(E,A,B,F) -> converttext(run(C,V6,V7,D),D,V8) = converttext(run(E,V6,V9,F),F,V8)	Assumption define_determinism
2	accurately_translates(A,B,C,D,E,F) -> exactly_correspond(compile(C,A,D,E,F),C,B,F)	Assumption define_exactly_correspond

3	(all A accurately_translates(cT,lsP,sP,A,e1,e2))	Assumption cT_compiles_sP
4	cA = stage2	Goal always_equal
5	deterministic(sP,lsP)	Assumption sP_deterministic
6	-deterministic(A,B) | -portable(A,B) | -exactly_correspond(C,A,B,D) | -exactly_correspond(E,A,B,F) | converttext(run(E,V6,V7,F),F,V8) = converttext(run(C,V6,V9,D),D,V8)	Clausify 1
7	accurately_translates(cT,lsP,sP,A,e1,e2)	Clausify 3
8	-accurately_translates(A,B,C,D,E,F) | exactly_correspond(compile(C,A,D,E,F),C,B,F)	Clausify 2
9	exactly_correspond(cP,sP,lsP,eA)	Assumption cP_corresponds_to_sP
10	portable(sP,lsP)	Assumption sP_portable
11	compile(A,B,C,D,E) = extract(converttext(run(B,retarget(A,E),C,D),D,E))	Assumption define_compile
12	stage1 = compile(sP,cT,e1effects,e1,e2)	Assumption definition_stage1
13	stage1 = extract(converttext(run(cT,retarget(sP,e2),e1effects,e1),e1,e2))	Para 11 12
14	extract(converttext(run(cT,retarget(sP,e2),e1effects,e1),e1,e2)) = stage1	Copy 13, flip
15	stage2 = compile(sA,stage1,e2effects,e2,eArun)	Assumption definition_stage2
16	stage2 = extract(converttext(run(stage1,retarget(sA,eArun),e2effects,e2),e2,eArun))	Para 11 15
17	cA = compile(sA,cP,eAeffects,eA,eArun)	Assumption definition_cA
18	cA = extract(converttext(run(cP,retarget(sA,eArun),eAeffects,eA),eA,eArun))	Para 11 17

19	cA != stage2	Deny 4
20	extract(converttext(run(cP,retarget(sA,eArun),eAeffects,eA),eA,eArun)) != stage2	Para 18 19
21	extract(converttext(run(cP,retarget(sA,eArun),eAeffects,eA),eA,eArun)) != extract(converttext(run(stage1,retarget(sA,eArun),e2effects,e2),e2,eArun))	Para 16 20
22	extract(converttext(run(stage1,retarget(sA,eArun),e2effects,e2),e2,eArun)) != extract(converttext(run(cP,retarget(sA,eArun),eAeffects,eA),eA,eArun))	Copy 21, flip
23	-portable(sP,lsP) | -exactly_correspond(A,sP,lsP,B) | -exactly_correspond(C,sP,lsP,D) | converttext(run(C,E,F,D),D,V6) = converttext(run(A,E,V7,B),B,V6)	Resolve 5 6
24	exactly_correspond(compile(sP,cT,A,e1,e2),sP,lsP,e2)	Resolve 7 8
25	exactly_correspond(extract(converttext(run(cT,retarget(sP,e2),A,e1),e1,e2)),sP,lsP,e2)	Para 11 24
26	exactly_correspond(stage1,sP,lsP,e2)	Para 14 25
27	-exactly_correspond(A,sP,lsP,B) | -exactly_correspond(C,sP,lsP,D) | converttext(run(C,E,F,D),D,V6) = converttext(run(A,E,V7,B),B,V6)	Resolve 23 10
28	-exactly_correspond(A,sP,lsP,B) | converttext(run(A,C,D,B),B,E) = converttext(run(cP,C,F,eA),eA,E)	Resolve 27 9
29	converttext(run(stage1,A,B,e2),e2,C) = converttext(run(cP,A,D,eA),eA,C)	Resolve 28 26
30	compile(A,stage1,B,e2,C) = extract(converttext(run(cP,retarget(A,C),D,eA),eA,C))	Para 29 11
31	extract(converttext(run(stage1,retarget(A,B),C,e2),e2,B)) = extract(converttext(run(cP,retarget(A,B),D,eA),eA,B))	Para 11 30
32	$F	Resolve 31 22

5.7.17 Discussion of proof #2

Note that proof #2's goal *could* be true, even if some of proof #2's assumptions (above) are false. First, note that the goal of proof #2 is:

```
stage2 = cA.
```

This equality *could*, in theory, have occurred by other means. As an extreme example, perhaps c_A was created by randomly generating data of the same length and then using it as an executable. In practice, even minor changes (other than changing comments) that invalidate any of proof #2's assumptions will tend to make this goal fail. As shown in section 7, DDC is extremely sensitive to even very minor deviations that make one of proof #2's assumptions false.

Since this has been proved, if c_A and stage2 are *not* equal, then at least one of the assumptions of proof #2 *must* be false. For example, if the compiler executable c_P is corrupted, then the assumption cP_exactly_corresponds is no longer true, and that could lead to c_A and stage2 being unequal. Similarly, if the compiler executable c_A is corrupted (e.g., it was replaced by some corrupt executable), then the assumption definition_cA is no longer true. If any failure of an assumption produces a different c_A, then c_A and stage2 will no longer be the same. Unfortunately, if we only know that c_A and stage2 are unequal, we cannot determine simply from this proof *which* assumption(s) are false. But at least we can be confident that, if they are unequal, at least one of these assumptions is false; we can then try to obtain other information to determine the cause(s).

Note that this proof permits $s_P \neq s_A$ and $c_P \neq c_A$, but it does not *require* it. Thus, it's quite possible that $s_P = s_A$ and/or $c_P = c_A$.

5.8 Proof #3: Goal cP_corresponds_to_sP

Proof #2 is intentionally designed to not require that a "grandparent" compiler exist in the putative origins of c_A. But having a grandparent compilers is a common circumstance,

and in this circumstance, one of the assumptions of proof #2 can be proved using other assumptions that may be easier to confirm.

Proof #2 depended on assumption cP_corresponds_to_sP (see section 5.7.12):

```
exactly_correspond(cP, sP, lsP, eA).
```

If a putative grandparent compiler c_{GP} does exist, this assumption is easily proven given some different assumptions. Simply reuse define_exactly_correspond as already defined, and add definition definition_cP and assumption cGP_compiles_sP as described below.

5.8.1 Definition definition_cP

First, we must define how cP was putatively generated – by grandparent compiler c_{GP}:

```
cP = compile(sP, cGP, ePeffects, eP, eA).
```

Note the strong similarity to definition_cA used earlier in section 5.7.14.

5.8.2 Assumption cGP_compiles_sP

We also need to assume that the grandparent compiler cGP will accurately translate the source code s_P:

```
all EnvEffects accurately_translates(cGP, lsP, sP, EnvEffects, eP, eA).
```

Note the strong similarity to cT_compiles_sP in section 5.6.4.

5.8.3 Goal cP_corresponds_to_sP

Given define_exactly_correspond, definition_cP, and cGP_compiles_sP, as described above, the goal is trivially proved by prover9 (as shown below). Recall that the goal is:

```
exactly_correspond(cP, sP, lsP, eA).
```

5.8.4 Prover9 proof of cP_corresponds_to_sP

Table 4 presents the proof found by prover9:

Table 4: Proof #3 (cP_corresponds_to_sP) in prover9 format

#	Formula	Rationale
1	(all A accurately_translates(cGP,lsP,sP,A,eP,eA))	Assumption cGP_compiles_sP
2	accurately_translates(A,B,C,D,E,F) -> exactly_correspond(compile(C,A,D,E,F),C,B,F)	Assumption define_exactly_correspond
3	exactly_correspond(cP,sP,lsP,eA)	Goal cP_corresponds_to_sP
4	-accurately_translates(A,B,C,D,E,F) | exactly_correspond(compile(C,A,D,E,F),C,B,F)	Clausify 2
5	accurately_translates(cGP,lsP,sP,A,eP,eA)	Clausify 1
6	cP = compile(sP,cGP,ePeffects,eP,eA)	Assumption definition_cP
7	-exactly_correspond(cP,sP,lsP,eA)	Deny 3
8	-exactly_correspond(compile(sP,cGP,ePeffects,eP,eA), sP,lsP,eA)	Para 6 7
9	exactly_correspond(compile(sP,cGP,A,eP,eA),sP,lsP,eA)	Resolve 4 5
10	$F	Resolve 9 8

5.8.5 Discussion of proof #3

Again, note that proof #3's goal is:

```
exactly_correspond(cP, sP, lsP, eA).
```

As noted earlier, this correspondence could have occurred by other means than by using a grandparent compiler (e.g., perhaps c_P was created by a human). In these cases, we may be able to perform other activities that give us confidence that this goal is true, and then use that information to justify the use of proof #2.

6 Methods to increase diversity

DDC must be executed as a "trusted" process, using a "trusted" compiler c_T and "trusted" environment(s) e1 and e2. As previously noted, in this dissertation "trusted" simply means that there is a high degree of confidence that there are no triggers that apply or no payloads that have an effect on the actions performed during DDC (e.g., when compiling s_A or s_P). Confidence in the DDC result depends on confidence in the assumptions.

This confidence can be gained in a variety of ways. One method to gain such confidence is to perform a complete formal proof of the compiler executable c_T and of the environments used in DDC, along with evidence that what actually runs is what was proved. But such proofs are difficult to perform with compilers typically used in industry. Another method to gain such confidence is to re-apply DDC on compiler c_T and/or the DDC environments; this can help, but re-applying DDC would require the use of yet *another* trusted compiler and environments, and this application of DDC would repeat until there was (1) a "final" trusted compiler and environments, or (2) a loop of trusted compilers and environments. In either case, at that point some *other* method is needed to increase confidence in the trusted compiler and environments.

A simple method to gain such confidence is through diversity. Diversity can *greatly* reduce the likelihood that c_T and the DDC environments have relevant triggers and payloads, often at far less cost. There are many ways we can gain diversity; these include diversity in compiler implementation, in time, in environment, and in input source code. These can be combined to further increase confidence that relevant triggers and payloads will not activate.

6.1 Diversity in compiler implementation

Compiler c_T's executable should be a completely different implementation than compiler c_A or c_P. This means it would have no (or little) shared code, and no (or little) shared data structures. Using a completely different implementation reduces the risk that cT includes triggers or payloads that affect c_P or c_A. Compiler c_T's executable could include triggers and payloads for compilers other than c_T, but this is less likely.

Ideally, no previous version of compiler c_T would have been compiled by any version of compiler c_A or c_P, even in c_T's initial bootstrap. This is because compiler c_A or c_P could insert into the executable code some routines to check for any processing of compiler c_A or c_P so that it can later "re-infect" itself. This kind of attack is difficult to do, especially since bootstrapping is usually done very early in a compiler's development and an attacker may not even be aware of compiler c_T's existence. One of the most obvious locations where this might be practical might be in the I/O routines. However, I/O routines are more likely to be viewed at the assembly or machine level than some other

routines (e.g., to do performance analysis), so an attacker risks discovery if they subvert I/O routines.

6.2 Diversity in time

If compiler c_T and the DDC environment were developed long before the compiler c_P and c_A, and they do not share a common implementation heritage, it is improbable that compiler c_T or its environment would include relevant triggers for a not-yet-implemented compiler. Magdsick makes a similar point [Magdsick2003].

The reverse (using a newer compiler executable to check an older compiler executable) gains less confidence. This is because it is easier for a recently-released compiler executable to include triggers and payloads for many older compilers, including completely different compilers. Nevertheless, this can still increase confidence somewhat, since to avoid detection by DDC the attacker must successfully subvert multiple compiler executables.

Diversity in time can only provide significant confidence if it can be clearly verified that the "older" materials are truly the ones that existed at the earlier time. This is because a resourceful attacker could tamper with those copies if given an opportunity to do so. Instead, protected copies of the original media should be preferred to reduce the risk of tampering. Multiple independently-maintained copies can be compared with each other to verify that the data used is correct. Cryptographic hashes can be used to verify the media; multiple hash algorithms should be used, in case a hash algorithm is broken.

An older executable version of compiler c_A or c_P can be used as compiler c_T, if there is reason to believe that the old version is not corrupt or that any Trojan horse in the old version of c_A will not be triggered by s_A. Note that this is a weaker test; the common ancestor could have been subverted. This technique gives greater confidence if the changes in the compiler have been so significant that the newer version is in essence a different compiler, but it would be best if compiler c_T were truly a separate implementation.

6.3 Diversity in environment

Different environments could be used in the DDC process than were used for the original generation of c_A. The term "environment" here means the entire infrastructure supporting the compiler including the CPU architecture, operating system, supporting libraries, and so on. Using a completely different environment counters Trojan horses whose triggers and payloads are actually in the executables of the environment, as well as countering triggers and payloads that only work on a specific operating system or CPU architecture.

These benefits could be partly achieved through emulation of a different system. There is always the risk that the emulation system or underlying environment could be subverted specifically to give misleading results, but attackers will often find this difficult to achieve, particularly if the emulation system is developed specifically for this test (an attacker might have to develop the attack before the system was built!).

In any case, the environment used to execute the DDC process should be isolated from other tasks. It should not be running any other processes (which might try to use kernel

vulnerabilities to detect a compilation and subvert it), and it should have limited (or no) network access.

6.4 Diversity in source code input

Another way to add diversity would be to use mutated source code [Draper1984] [McDermott1988]. The purpose of mutating source code is to make it less likely that triggers designed to attack the compilation of s_P or s_A will activate, and if they do, to reduce the likelihood that any payloads will be effective.

In terms of DDC, compiler c_T would become a source code transform (the mutator), a compiler (possibly an original compiler) c_X, and possibly a postprocessing step. These mutations could be implemented by automated tools, or even manually. The resulting c_T must be trusted, so trust must be given to the mutator(s), and the mutators must cause sufficient change so that any triggers or payloads in c_X will not have an effect when used as part of DDC.

There are two major types of mutations of source code: semantics-preserving and non-semantics preserving:

- In semantics-preserving mutations, the source code is changed to an equivalent program (that is, it will continue to produce the same outputs given the same inputs). This could include mutations such as renaming items (such as variables, functions, and/or filenames), reordering statements where the order is irrelevant, and regrouping statements. It can also include much more substantive changes, such as translating the source code into a different programming language. Even

trivial changes, such as changing whitespace, slightly increases diversity (though typically not enough by itself to justify a claim that all potential triggers and payloads are disabled). Forrest discusses several methods for introducing diversity [Forrest1997].

- In non-semantics-preserving mutations, the original semantics of the source code as presented to the compiler are *not* preserved. Instead, the goal is to preserve the necessary semantics of the source code when executed with the addition of preprocessing of its input to the execution and/or postprocessing of the execution output. Often this involves adding extraneous functionality to the source code, whose output is removed by the postprocessor, in the hope that this will cause triggers and payloads to fail. For example, the mutator may insert an additional text formatter that generates formatted output as well as an executable; the postprocessor must then remove or throw out that extraneous information. One challenge of this approach is that since semantics are no longer preserved, the postprocessing must remove changes that would affect DDC. McDermott discusses the advantage of this approach [McDermott1988].

Mutations can also be used to determine language lsP with greater precision. Presume that we have a non-mutated s_P and that we can verify c_A using DDC. We can then apply successive semantics-preserving mutations to s_P (e.g., focusing on areas that the language specification leaves undefined) and see if they cause a false negative. If a mutation causes a false negative, that mutation reveals an undocumented requirement of language lsP. (Credit goes to Aaron Hatcher, who made this observation.)

7 Demonstrations of DDC

The formal proof only shows that if something could be done, it would produce certain specific results. This chapter documents several demonstrations showing that DDC can be performed in the real world, and is thus a *practical* technique. This chapter presents results from tcc (a small C compiler), ported versions of Goerigk's Lisp compilers (one of which was known to be malicious), and the widely-used industrial-strength GNU Compiler Collection (GCC) C compiler. In some cases, it will be important track certain libraries separately from the "compiler source code" as it is traditionally defined; in such cases, the figures will show them as separate inputs.

7.1 tcc

Before [Wheeler2005], there had been no public evidence that DDC had been used. One 2004 GCC mailing list posting stated, "I'm not aware of any ongoing effort," [Lord2004]; another responded, "I guess we all sorta hope someone else is doing it." [Jendrissek2004]. This section describes its first demonstration (from [Wheeler2005]).

A public demonstration requires a compiler whose source code is publicly available. Other ideal traits for the initial test case included being relatively small and self-contained, running quickly (so that test runs would be rapid), having an open source

software license (so the experiment could be repeated and changes could be publicly redistributed [Wheeler2005]), and being easily compiled by another compiler. The compiler needed to be relatively defect-free, since defects would interfere with these tests. The Tiny C Compiler, abbreviated as TinyCC or tcc, was chosen as it appeared to meet these criteria.

The compiler tcc was developed by Fabrice Bellard and is available from its website at http://www.tinycc.org/. This project began as the Obfuscated Tiny C Compiler (OTCC), a very small C compiler Bellard wrote to win the International Obfuscated C Code Contest (IOCCC) in 2002. He then expanded this small compiler so that it now supports all of ANSI C, most of the newer ISO C99 standard, and many GNU C extensions including inline assembly. The compiler tcc appeared to meet the requirements given above. In addition, tcc had been used to create "tccboot," a Linux distribution that first booted the compiler and then recompiled the entire kernel as part of its boot process. This capability to compile almost all code at boot time could be very useful for future related work, and suggested that the compiler was relatively defect-free.

The following sections describe the test configuration, the DDC process, problems with casting 8-bit values and long double constants, and final results.

7.1.1 Test configuration

All tests ran on an x86 system running Red Hat Fedora Core 3. This included Linux kernel version 2.6.11-1.14_FC3 and GCC version 3.4.3-22.fc3. GCC was both the

bootstrap compiler and the trusted compiler for this test; tcc was the simulated potentially corrupt compiler.

First, a traditional chain of recompilations was performed using tcc versions 0.9.20, 0.9.21, and 0.9.22. After bootstrapping, a compiler would be updated and used to compile itself. Their gzip compressed tar files have the following SHA-1 values (provided so others can repeat this experiment):

```
6db41cbfc90415b94f2e53c1a1e5db0ef8105eb8  0.9.20
19ef0fb67bbe57867a590d07126694547b27ef41  0.9.21
84100525696af2252e7f0073fd6a9fcc6b2de266  0.9.22
```

As is usual, any such sequence must start with some sort of bootstrap of the compiler. GCC was used to bootstrap tcc-0.9.20, causing a minor challenge: GCC 3.4.3 would not compile tcc-0.9.20 directly because GCC 3.4.3 added additional checks not present in older versions of GCC. In tcc-0.9.20, some functions are declared like this, using a GCC extension to C:

```
void *__bound_ptr_add(void *p, int offset) __attribute__((regparm(2)));
```

but the definitions of those functions in tcc's source code omit the __attribute__((regparm(...))). GCC 3.4.3 perceives this as inconsistent and will not accept it. Since this is only used by the initial bootstrap compiler, we can claim that the bootstrap compiler has two steps: a preprocessor that removes these regparm statements, and the regular GCC compiler. The regparm text is only an optimization with no semantic change, so this does not affect our result.

This process created a tcc version 0.9.22 executable file which we have good reasons to believe does not have any hidden code in the executable, so it can be used as a test case.

Now imagine an end-user with only this executable and the source code for tcc version 0.9.22. This user has no way to ensure that the compiler has not been tampered with (if it has been tampered with, then its executable will be different, but this hypothetical end-user has no "pristine" file to compare against). Would DDC correctly produce the same result?

7.1.2 Diverse double-compiling tcc

Real compilers are often divided into multiple pieces. Compiler tcc as used here has two parts: the main compiler (file tcc) and the compiler run-time library (file libtcc1.a; tcc sometimes copies portions of this into its results). For purposes of this demonstration, these were the only components being checked; everything else was assumed to be trustworthy for this simple test (this assumption could be removed with more effort). The executable file tcc is generated from the source file tcc.c and other files; this set is notated s_{tcc}. Note: the tcc package also includes a file called tcclib, which is not the same as libtcc1.

Figure 2 shows the process used to perform DDC with compiler tcc. First, a self-regeneration test was performed to make sure we could regenerate files tcc and libtcc1; this was successful. Then DDC was performed. Notice that stages one and two, which are notionally one compilation each, are actually two compilations each when applied to compiler tcc because we must handle two components in each stage (in particular, we must create the recompiled run-time before running a program that uses it).

One challenge is that the run-time code is used as an archive format (".a" format), and this format includes a compilation timestamp of each component. These timestamps will, of course, be different from any originals unless special efforts are made. Happily, the runtime code is first compiled into an ELF .o format (which does not include these timestamps), and then transformed into an archive format using a trusted program (ar). So, for testing purposes, the libtcc1.o files were compared and not the libtcc1.a files.

Unfortunately, when this process was first tried, the DDC result did not match the result from the chain of updates, even when only using formats that did not include compilation timestamps. After much effort this was tracked to two problems: a compiler defect in sign-extending values cast to 8-bit values, and uninitialized data used while storing long double constants. Each of these issues is discussed next, followed by the results after resolving them.

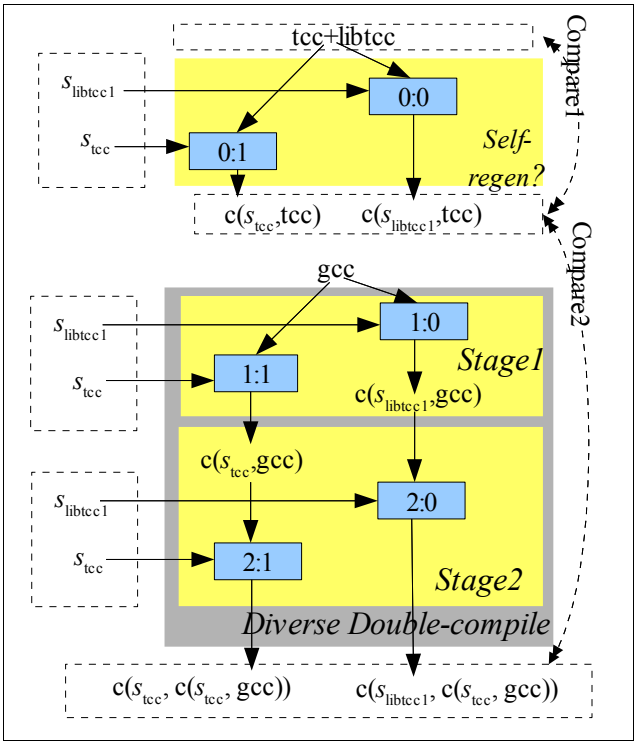

Figure 5: Diverse double-compiling with self-regeneration check, using tcc

7.1.3 Defect in sign-extending cast 8-bit values

A subtle defect in tcc caused serious problems. The defect occurs if a 32-bit unsigned value is cast to a signed 8-bit value, and then that result is compared to a 32-bit unsigned value without first storing the result in a variable (which should sign-extend the 8-bit value). Here is a brief description of why this construct is used, why it is a defect, and the impact of this defect.

The x86 processor machine instructions can store 4 byte constants as 4 bytes, but since many such constants are in the range -128..127, constants in this range can also be stored in a shorter 1-byte format (by specifying a specific ModR/M value in the machine

instruction). Where possible, tcc tries to use the shorter form, using statements like this to detect them (where e.v is of type uint32, an unsigned 32-bit value):

```
if (op->e.v == (int8_t)op->e.v && !op->e.sym) {
```

Unfortunately, the value cast to (int8_t) is not sign-extended by tcc version 0.9.22 when compared to an unsigned 32-bit integer. Version 0.9.22 does drop the upper 24 bits on the first cast to the 8-bit signed integer, but it fails to sign-extend the remaining 8-bit signed value unless the 8-bit value is first stored in a variable. This is a defect, at least because tcc's source code depends on a drop with sign-extension and tcc is supposed to be self-hosting. It is even more obvious that this is a defect because using a temporary variable to store the intermediate result *does* enable sign-extension. Besides, this is documented as a known defect in tcc 0.9.22's own TODO documentation, though this was only discovered after laboriously tracking down the problem. According to Kernighan [Kernighan1998] section A6.2 and the ISO/IEC C99 standard section 6.3.1.3 [ISO1999], converting to a smaller signed type is implementation-defined, but conversion of that to a larger unsigned value should sign-extend. Note that GCC does do the drop and sign-extension (as tcc's author expects).

This defect results in incorrect code being generated by tcc 0.9.22 if it is given values in the range 0x80..0xff in this construct. But when compiling itself, tcc is lucky and merely generates slightly longer code than necessary in certain cases. Thus, a GCC-compiled tcc generates code of this form (where 3-byte codes are used) when compiling some inline assembly in the tcc runtime library libtcc1:

```
1b5: 2b 4d dc  sub 0xffffffdc(%ebp),%ecx
1b8: 1b 45 d8  sbb 0xffffffd8(%ebp),%eax
```

But a tcc-compiled tcc incorrectly chooses the "long" form of the same instructions (which have the same effect—note the identical disassembly):

```
1b5: 2b 8d dc ff ff ff  sub 0xffffffdc(%ebp),%ecx
1bb: 1b 85 d8 ff ff ff  sbb 0xffffffd8(%ebp),%eax
```

One of the key assumptions in DDC is that the two compilers agree on the semantics of the language being compiled. This tcc defect violates this assumption, causing the files to unexpectedly differ. To resolve this, tcc was modified slightly so it would store such intermediate values in a temporary variable, avoiding the defect; a better long-term solution would be to fix the defect.

This example shows that DDC can be a good test for unintentional compiler defects—small defects that might not be noticed elsewhere may immediately surface!

7.1.4 Long double constant problem

Another problem resulted from how tcc outputs long double constants. The tcc outputs floating point constants in the "data" section, but when tcc compiles itself, the tcc.c line:

```
if (f2 == 0.0) {
```

outputs inconsistent data section values to represent 0.0. The tcc compiled by GCC stores 11 0x00 bytes followed by 0xc9, while tcc compiled by itself generates 12 0x00 bytes. Because f2 has type "long double," tcc eventually stores this 0.0 in memory as a long double value. The problem is that tcc's "long double" uses only 10 bytes, but it is stored in 12 bytes, and tcc's source code does not initialize the extra 2 bytes. The two excess "junk" bytes end up depending on the underlying environment, causing variations in the output [Dodge2005]. In normal operation these bytes are ignored and thus cause no problems.

To resolve this, the value "0.0" was replaced with the expression (f1-f1), since f1 is a long double variable known to have a finite value there (e.g., it is not a NaN). This is semantically the same and eliminated the problem. A better long-term solution for tcc would be to always set these "excess" values to constants (such as 0x00).

7.1.5 Final results with tcc demonstration

After patching tcc 0.9.22 as described above, and running it through the processes described above, exactly the same files were produced through the chain of updates and through DDC. This is shown by these SHA-1 hash values for the compiler and its runtime library, which were identical for both processes:

```
c1ec831ae153bf33bff3df3c248b12938960a5b6 tcc
794841efe4aad6e25f6dee89d4b2d0224c22389b libtcc1.o
```

But can we say anything about unpatched tcc 0.9.22? We can, once we realize that we can (for test purposes) pretend that the patched version came first, and that we then

applied changes to create the unpatched version. Since we have shown that the patched version's source accurately represents the executable identified above, we only need to examine the effects of a reversed change that "creates" the unpatched version. Visual inspection of the reversed change quickly shows that it has no malicious triggers and payloads. Thus, we can add one more chain from the trusted compiler to a "new" version of the compiler that is the untouched tcc-0.9.22. Because of the changes in semantics and the flow of data, to get a stable result we end up needing to recompile several times. In the end, the following SHA-1 hash values are the correct executables for tcc-0.9.22 on an x86 in this environment when tcc is self-compiled a sufficient number of times to become "stable":

```
d530cee305fdc7aed8edf7903d80a33b6b3ee1db tcc
42c1a134e11655a3c1ca9846abc70b9c82013590 libtcc1.o
```

7.2 Goerigk Lisp compilers

A second demonstration of DDC using a small compiler was performed using a pair of Lisp compilers developed in [Goerigk2000] and [Goerigk2002]. This demonstrated that DDC can be applied to languages other than C, and that it can detect malicious compilers.

Goerigk developed both "correct" and "incorrect" compilers (Goerigk's terminology) using ACL2, a theorem-prover supporting a Common-Lisp-like language. Goerigk also developed an abstract machine simulator to run the code produced by the compilers. Using DDC on this pair of compilers demonstrates (1) the ability of DDC to detect a malicious compiler, including the differences in the malicious compiler, (2) reconfirm the

ability of DDC to detect the correct compiler executable, and (3) that DDC does not require C; these compilers are written in, and support, a LISP-based language.

To perform this demonstration, the compilers and virtual machine implementation originally written by Goerigk were first ported to Common Lisp. The compilers were originally written in ACL2, which is similar but not identical to Common Lisp. There are far more Common Lisp implementations than ACL2 implementations, so porting it to Common Lisp enabled the use of many alternative compilers. This port required removing uses of "defthm" (define theorem) and mutual recursion declarations (ACL2 requires all mutually-recursive functions to be specially declared; Common Lisp has no such requirement). A few ACL2-unique functions were rewritten in Common Lisp, to allow the existing code to run: LEN, ZP (returns true if parameter X is not an integer, or if X is integer and X=0), TRUE-LISTP (returns True if its argument is a list that ends in, or equals, nil), and ACL2-NUMBERP (is value a number). In addition, the "execute" command was renamed because on some Common Lisp implementations that is a predefined function name. The GNU Clisp implementation was then used to run the tests, though any Common Lisp implementation would have served.

As expected, both the correct and incorrect compilers would produce correct code for a simple sample program (in this case, for a factorial function). Both could regenerate themselves using the correct compiler source code as input, demonstrating that they could pass the compiler bootstrap test and the self-regeneration test. However, when given a

special “login” program, the compiler executables would produce *different* answers. Thus, these programs really do demonstrate the attack.

The DDC technique was then applied. First, it was applied to the correct source code, using the underlying Common Lisp implementation (clisp) as the trusted compiler c_T. The stage 2 output was then compared to the correct compiler executable, and was shown to be equal. The stage 2 output was then compared to the incorrect compiler executable, and was shown to be not equal. A unified diff was then applied to the stage 2 and incorrect compiler executable; this showed the “unexpected” differences, and immediately revealed that the difference had something to do with the login program. This difference is an immediate tip-off that there is something malicious happening; no compiler should be specifically looking for the login program, and then acting differently! An examination of the difference quickly revealed that it was comparing the login program, and then inserting different code in this special case.

Appendix A includes more detail, including the actual “diff” from machine file produced by DDC with the machine file of the incorrect compiler.

7.3 GCC

To conclusively demonstrate that DDC can be applied to “industrial-scale” compilers widely used in commercial applications, the DDC process was successfully applied to the GNU Compiler Collection (GCC), specifically the C compiler of GCC.

In 1983, Richard Stallman began searching for a compiler that would help meet his goal to create an entire operating system that could be viewed, modified, and redistributed (without limitations like royalties). He did not find an existing compiler that met his licensing, functionality, and performance requirements, so he began writing a C compiler from scratch. This became GCC. Today, GCC is a GNU Project directed by the Free Software Foundation (FSF), licensed under the GNU General Public License (GPL).

GCC is widely used, though specific statistics are difficult to find. "GCC's user base is large and varied... no direct estimate of the total number of GCC users is possible... [but] GCC is the standard compiler shipped in every major and most minor Linux distributions [and is] the compiler of choice for the various BSD operating systems... The academic computing community represents another large part of GCC's user base... GCC is also widely used by nonacademic customers of hardware and operating system vendors... [considering] the broad range of hardware to which GCC has been ported, it becomes quite clear that GCC's user base is composed of the broadest imaginable range of computer users." [vonHagen2006]

7.3.1 Setup for GCC

DDC can be used to regenerate an existing compiler executable, given enough information on how it was compiled and the other assumptions already discussed. However, after many fruitless attempts to do this with Fedora Core, it was found that the Fedora project (and probably many other distributions) does not record all the information necessary to easily recreate the exact same compiler executable from scratch.

In some cases there were dependencies on software that was not shipped with the distribution. This may seem surprising, but in practice this information has not been needed; many organizations record these files for later use instead of regenerating them.[10]

So for purposes of the experiment, a new GCC executable was created specifically to demonstrate DDC, using the publicly-available GCC source code. The executable was created using the GCC executable that comes with Fedora (which was a different version than the source code being compiled) as the "grandparent" compiler. To simplify the test, the compiler was self-regenerated, that is, s_P=s_A. The resulting compiler executable, after two compilation stages, was then considered to be the compiler-under-test c_A. Then, the DDC process was used (with a different trusted compiler) to determine if it would produce the same result as the compiler under test. This way, all necessary information for the experiment would be available.

The GCC suite includes a large number of different compilers for different languages. Attempting to cover all of these languages was not necessary for purposes of this dissertation. Thus, work focused on the C compiler. Future work could add support for other languages using the approach described here.

The GCC suite depends on a great deal of external software. This includes a linker (typically named "ld"), assembler (typically named "as"), archiver ("ar"), symbol table constructor ("ranlib"), and standard C library, as well as an operating system (especially a

[10] My thanks to Aaron Hatcher, who attempted to apply DDC to various versions of GCC included in Fedora Core, and to Jakub Jelinek of Red Hat, who tried to provide Aaron with the necessary information to regenerate the executables after-the-fact. Aaron's efforts were unsuccessful at the time, but they provided insight that later led to the successful application by Wheeler that is described here.

kernel) to run on. In particular, the C compiler cc1 generates assembly code, which is then assembled. For purposes of this experiment, all of these external programs were considered to be external to the compiler. These additional programs could have been covered by DDC by considering them as part of the compiler, however, doing so would have made this first experiment even more difficult, and would not have shown anything substantial. These other programs are not trivial, but the main C compiler is key; once we can show that DDC can handle the "real" C compiler, expanding the scope of DDC to cover these other programs (if desired) is merely a matter of additional effort.

To demonstrate DDC, a second trusted compiler was needed, one that was able to correctly process the large and complex GCC source code. After examining several compilers, the Intel C++ Compiler (icc) was chosen. In spite of its name, icc also includes a C compiler. Initial tests suggested that icc was a relatively reliable compiler, and icc supports many GCC extensions and implementation-defined behavior with the same semantics, making it more likely to successfully compile GCC. The latest version of icc available at the time, version 11.0, was used.

There are many different versions of GCC available, and for purposes of the experiment, any version of GCC would do as the compiler under test. However, it must be possible for the trusted compiler to compile the source code of the parent (in this case, it is the same as the compiler under test). The parent must also be able to compile the compiler under test (in this case, the compiler under test must be able to recompile itself). The newer GCC versions 3.4.4, 4.0.4, and 4.1.2 could not be easily recompiled by icc (giving

error messages instead), so they were not used for this experiment. Should DDC become a common process, compiler developers should test their compilers to ensure that they are easily compiled by *other* compilers. Remarkably, the source code for GCC version 3.1.1 could not be compiled by the GCC version installed in Fedora (version 4.3). For purposes of this experiment, GCC version 3.0.4 was selected to be the source code for the compiler under test, since it met these requirements.

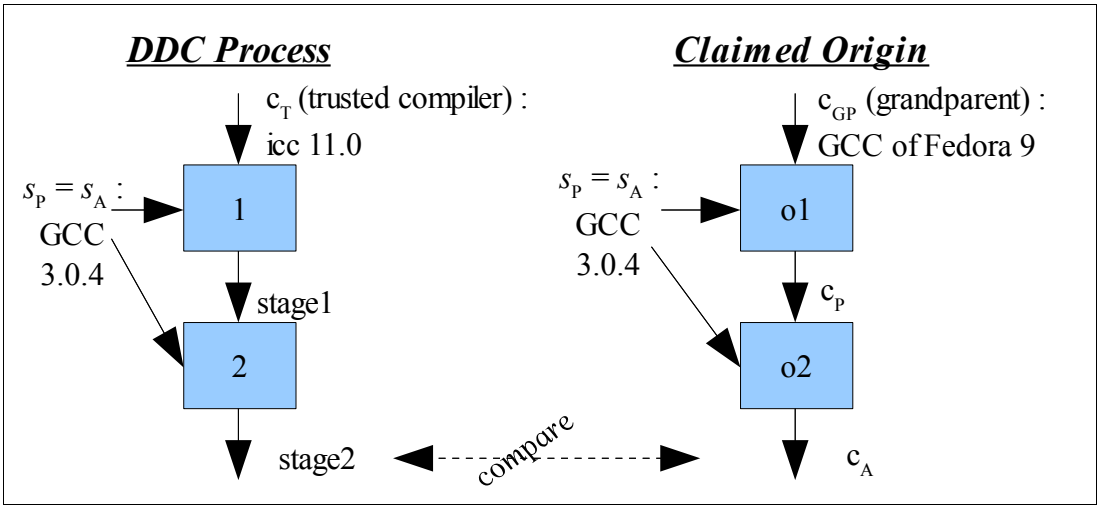

Figure 6: DDC applied to GCC

All compilations were performed on a personal computer running the Fedora 9 Linux distribution in 32-bit mode on an x86 system. Compiler caches were completely disabled at all times (by removing the package ccache), to ensure that all recompilations were actually performed. The "kernel-headers" package was also installed, since it defined key constants necessary for recompilation of GCC.

When recompiling the GCC compiler, a number of options are available, which unless required were held to their defaults. In particular, the "prefix" value, which identifies the prefix of its pathname when installed, was left as "/usr/local". Typical "final" installations often change this to "/usr"; leaving this value as-is reduced the risk that some intermediate step would attempt to overwrite the GCC compiler already included in the Fedora installation.

As with tcc, the recompilation of gcc had many sub-steps. In particular, certain run-time libraries were compiled first, before the compilation of the "main" compiler itself, just as with tcc.

7.3.2 Challenges

One piece of critical information that had to be recorded is the full pathname of the "master result" directory that contains the source code and object directories. This full pathname is actually embedded into the final executables, and so it must be the same in both the process used to create the compiler-under-test and in the DDC process. In terms of the DDC process, this pathname is one of the inputs to the DDC process. In the case of the experiment, this value was "/home/dwheeler/thesis/work", though the actual value was not really important – it merely had to be the same when creating the compiler-under-test and when performing DDC. The solution used in the experiment was to use exactly the same pathname when creating the compiler-under-test and when performing the DDC process.

The build process for the chosen version of GCC (3.0.4), as part of its “make compare” step, uses an obsolete format for the "tail" command. For example, it uses “tail +16c” to skip first 16 characters. By default this format is no longer accepted by modern GNU implementations of “tail”, which interpret “tail +16c” as an attempt to read from a file named “+16c”. This was resolved by setting the environment variable “_POSIX2_VERSION” to “199209” before the build is performed; GNU tail will notice that this environment variable is set and use the older (GCC-expected) semantics.

Unfortunately, even after these changes were made, the DDC process did not produce an executable equal to the compiler-under-test. This meant that one of its assumptions was not true. Determining why this was so (by tracking this backward through the executables and object code in a large compiler to determine the cause) was extremely time-consuming, due in part to the large size of GCC, and produced a very unexpected result. It turned out that GCC 3.0.4 did *not* fully rebuild itself when later build stages were requested, even though the GCC recompilation documents stated that they did, due to its “libiberty” run-time library routines.

The GCC compiler documentation explains that its normal full build process, called a “bootstrap”, can be broken into “stages”. The command “make bootstrap” is supposed to build GCC three times—once with the native compiler, once with the native-built compiler it just built, and once with the compiler it built the second time. Each step of this process is called a “stage”. [GNU2002, section 14]. The last two stages should produce the same results; “make compare” checks if this is true. This recompilation

process includes recompilation of the "libiberty" library, a collection of lower-level subroutines used by various GNU programs.

Unfortunately, actual GCC build behavior does not match the GCC documentation for "make bootstrap". The stage1 compiler was *not* used to recompile the internal libiberty library when creating stage2; instead, the results of stage1 were *directly copied* into stage2. This appears to be a side-effect of how the makefiles were written; when stage2 was performed, the make program determined that the libiberty object file was dated after the source, and skipped rebuilding it. Because of this, the resulting executable was actually a hodgepodge that combined the results of two *different* compilers into a single executable. After a long effort to track down this problem, it was noted that there was a hint about this defect in the GCC documentation, though its significance was not at all obvious at the time: "Libiberty [is only] built twice... fixing this, so that libiberty is built three times, has long been on the to-do list." [GNU2002, section 14]

It would be possible, though nontrivial, to directly apply DDC to this circumstance. In this case, we have a "parent" compiler that is different than the compiler under test, so we would require the source code for both the compiler under test and the parent compiler. But this would be a complex approach, far more complex than necessary for use as a real-world demonstration, and it was clear from the documentation that the *intent* of the compiler authors was to completely regenerate the compiler in stage2.

Instead, the GCC Makefile was modified to permit finer control over the building process. Then the process to rebuild the compiler (for both the compiler under test and DDC) was modified so it correctly recompiled the entire compiler in stage 2, by doing:

- "make all-bootstrap", which used the "initial" compiler to compile libraries (such as libiberty) and necessary bootstrap tools to prepare for stage1. The "initial" compiler for the "compiler under test" was a different version of GCC. The initial compiler for DDC was, instead, icc.
- "make stage1_build" to build the first stage GCC.
- A forced rebuild of libiberty, using the new stage1 compiler.
- "make stage2_build" to produce the final stage2 GCC.
- Although not strictly necessary, a "make stage3_build" followed by "make compare" was also done to detect certain kinds of recompilation errors.

7.3.3 GCC Results

Once the corrected GCC build process was used, DDC produced bit-for-bit identical results with the compiler under test, as expected. The resulting GCC compiler is actually a set of files, instead of a single file. Appendix B presents the detailed results.

8 Practical challenges

There are many practical challenges to implementing this DDC technique. This chapter discusses some of these challenges and how to overcome them. Some of this information was discovered or extended through the process of implementing the demonstrations.

8.1 Limitations

All techniques have limitations. DDC only shows that a particular executable corresponds to a particular source code, resulting in these key limitations of the DDC:

1. There may be other executables that contain Trojan horse(s) and yet claim to correspond to a given source. This can be resolved by using cryptographic hashes of the executable and the source code, and including their hashes when reporting that DDC succeeds.
2. The source code may have malicious code (such as Trojan horses) and/or errors, in which case the executable file will too. However, if the source and executable correspond, the source code can be analyzed in the usual ways to find such problems. Thus, DDC does not eliminate the need for review; instead, it allows review processes to concentrate on the source code, knowing that if certain other assumptions hold, DDC can show that the executable will correspond to the

source code. In short, DDC can show that there is "nothing hidden", enabling review of source code instead of executable code.

3. When the DDC result is not equal to the original compiler under test, there are many potential causes. If they are unequal, at least one of the assumptions of proof #2 has been violated, but it may not be apparent which assumption(s) have been violated. Determining the cause may require examining differences of executables and/or the compilation process, which for large compilers can be difficult and time-consuming. If a compiler executable does not correspond with its source code, it is corrupted, but this corruption need not be malicious. However, as shown in appendix A, it is sometimes possible to examine the differences and determine that the corruption is malicious.

8.2 Non-determinism

Uncontrolled non-determinism may cause a compiler to generate different answers for the same source input. Even uninitialized values can create this non-determinism, as was the case for tcc (see section 7.1.4). It may be easiest to modify the compiler so that it can be made to be deterministic (e.g., add an option to set a random number seed and never store uninitialized data in a resulting executable).

Differences that do not affect the outcome do not affect DDC. For example, heap memory allocations during compilation often allocate different memory addresses between executions, but this is only a problem if the compiler output changes depending on the specific values of the addresses. Roskind reports that variance in heap address

locations affected the output of at least some versions of the Javasoft javac compiler. He also stated that he felt that this was a bug, noting that this behavior made port validation extremely difficult [Roskind 1988]. Most compilers avoid such non-determinism because it makes testing difficult.

8.3 Difficulty in finding alternative compilers

It may be difficult to compile s_A or s_P using other existing compilers. There may not be any other compilers for the general language used to write s_A or s_P. Alternatively, s_A or s_P may use non-portable extensions. Yet DDC requires a trusted compiler!

Thankfully, there are many possible solutions if s_A or s_P cannot be compiled by existing compilers. The DDC technique only requires that a compiler with the necessary properties be created. An existing compiler could be modified (e.g., to add extensions) so it can perform the necessary compilation. Another alternative is to create a trusted preprocess step, possibly done by hand; in this case c_T would be defined as being the preprocess step plus the existing compiler. It is also possible to write a new trusted compiler from scratch. Since performance of the trusted compiler is irrelevant, and it only needs to be able to compile one program, this may not be difficult.

It may be possible to use an older version of c_A as c_T, but that is far less diverse so the results are far less convincing. Doing so also risks "pop-up" attacks, described next.

8.4 Countering "pop-up" attacks

A "pop-up" attack, as defined in this paper, is where an attacker includes a self-perpetuating attack in only *some* versions of the source code (where the attack "pops up"), and not in others. The attacker may choose to do this if, for example, the attacker believes that defenders only examine the source code of some versions and not others.

Imagine that some trusted compiler c_T is used to determine that an old version of compiler c_A—call it c_{A1}— corresponds to its source s_{A1}. Now imagine that an attacker cannot modify executables directly (e.g., because they are regenerated in a separate controlled process), but that the attacker can modify the source code of the compiler (e.g., by breaking into its repository). The attacker could sneak malevolent self-perpetuating code into s_{A2} (which is used to generate c_{A2}), and then remove that malevolent code from s_{A3}. If c_{A2} is used to generate c_{A3}, then c_{A3} may be malicious, even though s_{A3} does not contain malevolent code and c_{A1} corresponded to s_{A1}. Examination of every change in the source code at each stage can prevent this, but this must be thorough; examining only the source's beginning and end-state will miss the attack.

The safest way to counter "pop-up" attacks is to re-run DDC on every executable release before the executable is used as a compiler, using a trusted compiler c_T. If that is impractical, at least use DDC periodically and unpredictably to reduce the attack window and increase the attacker's risk of discovery.

8.5 Multiple subcomponents

Compilers may have multiple subcomponents (such as a preprocessor, a front end, a back end, a peephole optimizer, a linker, a loader, and one or more runtime libraries). All of these subcomponents could be in different files and generated by separate recompilation steps. If these recompilations can be done in any order, and there is no interaction between them, we can simply perform each step, in any order. But if compiling a subcomponent depends on the result of recompiling another subcomponent (e.g., because it's a runtime library that will be embedded in the resulting executable), then these dependencies must be honored, just as when recompiling the compiler for any other reason. In general, if the order of internal steps matters during compilation of s_P and s_A, then DDC must use the same order as was used to create the original c_P and c_A.

Compiler c_T may have multiple components, but since its recompilation is out-of-scope of DDC, this is irrelevant. All that is necessary is that c_T have the required properties (as a suite) for DDC.

8.6 Inexact comparison

In certain cases, inexact comparisons may be needed. Comparisons need not require an identical result as long as it can be shown that the differences do not cause a change in behavior. This might occur if, for example, outputs included embedded compilation timestamps. However, showing that differences in files do not cause differences in the functionality, in the presence of an adversary, is extremely difficult. Another approach is

to first work to make the results identical, and then show that the steps leading from that trusted point do not introduce an attack.

8.7 Interpreters and recompilation dependency loops

In some cases, what is executed bears a more complicated relationship to source code than has been shown so far, but the trusting trust attack can still be countered using DDC.

It does not matter if the executable is a sequence of native machine code instructions or something else (such as an "object file", "byte code", or non-native instructions). All that is required is that there be some environment that can execute the instructions. If there is a concern that some parts of the environment may be corrupted, consider those parts as part of the compiler (this requires their source code) and apply DDC.

Many language implementations do not generate a separate executable that is run later. They may read and immediately execute source code (call it s_E) a line at a time, or they may compile source code s_E to an executable (often a specialized byte code) each time the source code is run and not save the executable for later use. In these cases, the trusting trust attack does not directly apply to s_E, since there is no separate executable in which malicious code can be hidden. However, these language implementations tend to themselves be compiled executables (for speed). Any language implementations that are compiled *are* vulnerable to the trusting trust attack, and DDC still applies to them.

As noted in section 4.4, DDC can be applied to compilers that recompile themselves (as a special case). When compilers do not recompile themselves, DDC can be repeatedly

applied to each ancestor compiler, from oldest to newest, to demonstrate that each of the ancestor compilers are not corrupt. If there is a "loop" of compilers (e.g., compiler c_A is used to compile compiler c_B, and c_B is used to compile the next version of compiler c_A), DDC can still be used; arbitrarily choose a compiler to check, and "break the loop" using an alternative trusted compiler.

8.8 Untrusted environments and broadening DDC application

The environment of c_A may be untrusted. As noted earlier, an attacker could place the trigger mechanism in the compiler's supporting infrastructure such as the operating system kernel, libraries, or privileged programs. Triggers would be especially easy to place in assemblers, linkers, and loaders. But even unprivileged programs might be enough to subvert compilations; an attacker could create a program that exploited unknown kernel vulnerabilities.

The DDC technique can be used to cover these cases as well. Simply redefine the "compiler" c_A to include the set of all components to be checked, and not just the traditional interpretation of the term "compiler". This could even include the set of all software that runs on that machine (including all software run at boot time), not just the compiler proper. The source code for all this software to be checked would still be termed s_A, but s_A would now be much larger. Consider obtaining c_A and s_A from some read-only medium (e.g., CD-ROM or inactive hard drive); do not trust this redefined untrusted c_A to produce itself (e.g., by copying c_A's files using c_A)! Then, using DDC on a different (trusted) environment to check c_A. Depending on the scope of this new c_A and

s_A, this might regenerate all of the operating system (including boot software), various application programs, and so on. If DDC can regenerate the original c_A, then the entire set of components included in c_A are represented by the entire set of source code in s_A. There is still a risk that s_A includes malicious code which is embedded in c_A; if c_A or its environment might have code that shrouds s_A (so that the s_A viewed is not the actual s_A), always use a separate trusted system to view or print s_A when reviewing s_A.

An alternative approach to countering potentially-malicious environments is to maximize the amount of software that is used in source code form, and boot a relatively small "compiler". This is already done with many "scripting" languages (such as typical implementations of Python and PHP). It can, however, also be done with languages that are typically compiled. The original developer of tcc demonstrated that the tcc C compiler could be booted with a relatively small infrastructure; it could then recompile the operating system (including the Linux kernel) and then run those results.

A resourceful attacker might attack the system performing DDC (e.g., over a network) to subvert its results. If this is a concern, DDC should be done on isolated system(s). Ideally, the systems used to implement DDC should be rebuilt from trustworthy media, not connected to external networks at all, and not run any programs other than those necessary for the test.

8.9 Trusted build agents

Few will want to do DDC themselves. Organization(s) trusted by many others (such as government agencies or trusted organizations sponsored by them) could perform DDC on

a variety of important compilers, as they are released, and report the cryptographic hash values of the executables and their corresponding source code. The source code would not need to be released to the world, so this technique even could be applied to proprietary software (though without the source code, the information that they correspond is much less useful). This would allow others to quickly check if the executables they received were, in fact, what their software developers intended to send. If someone did not trust those organizations, they could ask for another organization they did trust to do this, or do it themselves, if they can get the source code. Organizations that do checks like this have been termed "trusted build agents." [Mohring2004]

8.10 Application problems with current distributions

There are a number of "distributions" that combine open source software from a large variety of different origins, integrate them, and distribute the suite to end users. In theory, these should be easy to test using DDC. Efforts to recreate the GCC compiler distributed with Fedora, even with help from Red Hat, showed that this is not always easy.

Accurately re-creating a distribution's executable files requires extremely detailed and accurate information about compilation of the compiler. For example, recompiling GCC 3 with the same bit pattern requires knowing and reusing the full pathname of the directory used to store intermediate results. Distributors do not always record this detailed information; instead, they simply record the executable files (once created) in case they are needed again, instead of recording the information necessary to exactly recreate them. Some of this detailed information can be obtained by attempting to apply

DDC and examining the differences, e.g., compiling GCC with a different pathname for intermediate results, and comparing the results, will quickly reveal the original pathname. However, in some cases, the difference can be detected by DDC, but the cause of the difference may not be obvious.

One especially surprising finding was that obtaining the correct parent s_P can be difficult. Distributions typically release their software as a large set of interrelated "packages", and most distributions distribute precompiled executables of their packages. During development of a new distribution version, the compiler, libraries, and applications are all updated, sometimes multiple times. Once an executable (compiler or not) is created, it is frozen and tested. There is a strong incentive to *not* recompile the entire operating system when a compiler is revised, for if a problem occurs afterwards, it can be difficult to determine where the problem is. In contrast, if packages are recompiled and tested one at a time, then problems can be immediately pinpointed. As a result, the practice of incrementally testing and releasing executable files can easily lead to different packages being compiled by different versions of a compiler within the same distribution. If the compiler is modified several times during the distribution's release process, some packages may be compiled with a version of the compiler that is neither the previous released version nor the final released version version—but is an intermediate instead. What is more, compiler executables may incorporate material from other packages, which were themselves compiled with different versions of the compiler.

However, distributions could easily make minor modifications to their processes to make DDC easier to apply. Recording the information necessary to accurately reproduce an executable is one approach. Another approach is to freeze the compiler at an earlier stage, and recompile everything so the executables are stable (that is, they reproduce themselves precisely). Now that DDC has been demonstrated, compiler suppliers have more incentive to record the information necessary to recreate executables.

There are other issues with current Linux distributions that can be easily worked around for DDC, but can cause trouble for the unwary:

- Many Linux distributions use "prelink", which modifies the files of executable commands and libraries of a running system to speed their later invocation. This is not a problem as long as the files are captured and compared using DDC *before* they are adjusted by prelink.
- Many Linux distributions use "ccache", a system that caches compilation results and quickly replies with previous results if the inputs and compiler are "the same". If the caching system incorrectly determines that the compiler being invoked is "the same", but is in fact different, then the wrong results will be used if it is part of the DDC process.

8.11 Finding errors and maliciously misleading code

DDC simply shows that source code corresponds to executable code (given some assumptions). This is a significant advantage, since software developers are far more

likely to review source code than an executable. At the very least, developers must review some source code when when they are preparing to change it.

This does not make it trivial; it may be difficult to find intentional vulnerabilities in large and complex software. But it does tend to make it easier to find intentional vulnerabilities. In particular, errors can be detected and resolved by traditional means as discussed in section 2.3.

But is it enough to ensure that the source code and executable correspond? An attacker who can modify compiler source code could insert *maliciously misleading code*, that is, code that is designed to *appear* to be correct but actually does something malicious instead. The Obfuscated V contest [Horn2004], the Underhanded C contest [Binghamton2005], and the Linux kernel attack (discussed in section 2.5) all show that it is possible to write maliciously misleading code.

The good news is that these public examples also suggest that simple measures can counter many of them. Some examples use misleading formatting, which can be countered by using a "pretty printer" to reformat source code before review. Some examples exploit buffer overflows; using languages or tools that prevent buffer overflows prevents them. Some examples use widely-known "common mistakes" for the given programming language (e.g., mistaking "=" for "==" in C); training human reviewers and using tools to highlight or forbid "confusing" constructs tends to counter common mistakes. In the longer run, languages can be designed or modified to make common mistakes less likely. For example, Java was specifically designed to make certain

common errors in C impossible or less likely. Tools can be developed to search for maliciously misleading code, yet not released to potential developers of maliciously misleading code, making it difficult for attackers to be confident that their attacks will go undetected.

8.12 Hardware

DDC can be extended to hardware, but two key observations must be noted first:

1. What some people call "hardware" is actually software. BIOS files and microcode are still software, and thus they can be handled the same way as any other software (including using DDC as described in this paper).
2. When people discuss hardware subversion, they are typically worried about the more obvious subversion approaches. In other words, they are concerned about ways that a CPU chip or support chip could be directly modified to perform some malicious operation (such as to allow remote control or to include a shutoff date).

There *are* technical ways of countering attacks in the second item above, but they do not involve the use of DDC, because they do not involve subverting new hardware through hidden triggers and payloads in existing hardware. For example:

1. If the threat is that a human will insert malicious logic into the human-readable hardware design, then one approach is to review the designs (and make sure that what is used is what was reviewed).

2. If the threat is that a software program may insert malicious logic when it processes the hardware design, one approach is to review the software tool's source code. If its executable may have been tampered with, but the source code is fine and the generation process is trusted (e.g., the program does not regenerate itself), simply recompile the tool with the same circumstances as when it was last compiled and see if the resulting executable is identical.

3. If the threat is that tool output (e.g., the chip itself or a mask) may be subverted after it has left the tool, then if the tool can be made to be deterministic, rerunning that tool and comparing the "expected" results with the "actual" results should reveal any differences. In multi-step processes, rerun each step in sequence and determine if there is a difference. Doing this with hardware requires an "equality" operator for hardware components like masks and chips; as discussed below, determining if hardware is equal is often more difficult than for software.

There is another threat, however, that is rarely discussed: What if the computer hardware has been subverted so that *the hardware subverts the hardware development process of the next generation of hardware*? At this time, such attacks seem far less likely:

1. Hardware subversion of hardware's own development process is harder to do than for software. For software this kind of self-subversion is easy to do, because the attacking software is typically at a similar level of abstraction. In contrast, hardware is at such a different (lower) level of abstraction, making it more

difficult to create useful automated triggers and payloads in hardware that have a high probability of being useful in attacking the hardware design process for the next generation of hardware.

2. There is little need to implement such a complicated attack on hardware. There are many other difficult-to-counter attacks at the hardware level which are much easier to perform.

Still, if hardware self-subversion (where the hardware is designed to subvert the design of the next generation of the hardware) is considered a threat, then DDC *can* be used to counter it. However, there are some important challenges when applying DDC to hardware: One technical, and one legal.

First, the technical challenge: For DDC to work on hardware, it needs an "equality" operator. It may be that a scanning electron microscope, used with varying angles/positions, could gather enough information to determine if a chip was "equal to" another chip (real or virtual) with an acceptable level of probability, especially if it were supplemented with other test techniques that check electrical connectivity in a variety of locations. Note that DDC when applied to hardware can only show that one particular chip matches its design; another chip might not. That is true for the software approach as well, but checking for equality is much easier for software than for hardware.

Second, there is a legal challenge: The necessary information is often difficult to legally obtain. Large amounts of hardware data, including the actual layout of the chip, is often

kept proprietary from even the chip designers. DDC requires that the correct hardware be known, so that it can be compared to the real hardware. Software developers would typically find it unacceptable if they couldn't see the bytes that their compilers produced. In contrast, hardware chips are routinely modified in the many manufacturing steps in ways not disclosed to the chip designers. For example, the libraries that Verilog and VHDL design tools show their users are often not what are really used on chips. In addition, because of quantum mechanical effects, at smaller scales there are corrections that some companies will do to a chip's layouts/wiring that chip designers are forbidden (by contract) to see. In addition, many hardware components are built out of IP ("intellectual property") cores from various organizations worldwide, in which designers are forbidden by contract to see their logic. Many chip designers are unaware that what is actually on their chips is not exactly what they designed, possibly because many chip designers are not near the foundries (making it easier to fool the chip designers). Should the use of DDC become important for hardware, such information would need to be available.

9 Conclusions and ramifications

This dissertation has shown that the trusting trust attack can be countered. Before this work began, the trusting trust attack had almost become an axiom of computer security, since many believed a successful attack to be undetectable. Although others had posted the idea of DDC before this work began, it had only been described in a few sentences at most, and only in obscure places. DDC had not even been given a name when this work began! This work has explained DDC in detail, provided a formal proof (with formalized assumptions), and demonstrated its use (including with a widely-used C compiler).

The DDC technique has many strengths: it can be completely automated, applied to any compiled language (including common languages like C), and does not require the use of complex mathematical proof techniques. Second-source compilers and environments are desirable for other reasons, so they are often already available, and if not they are also relatively easy to create (since high performance is unnecessary). Some unintentional defects in either compiler are also detected by the technique. The technique can be easily expanded to cover all of the software running on a system (including the operating system kernel, bootstrap software, libraries, microcode, and so on) as long as its source code is available.

Passing the DDC test when the trusted compiler and environment is not proven is not a mathematical proof, but more like a legal one. The DDC technique assumes that the DDC process (including trusted compiler c_T and the environments) does not have triggers or payloads that apply to the source code being compiled. In most practical cases, this assumption will not be formally proved. However, the DDC test can be made as rigorous as desired by decreasing the likelihood (e.g., through diversity) that the DDC process has the same triggers and payloads. Multiple diverse DDC tests, using different trusted compilers, can strengthen the evidence even further. Thus, defender can easily make it extremely unlikely that an attacker could avoid detection by the DDC technique.

The DDC technique only shows that the source code corresponds with a given compiler's executable, i.e., that nothing is hidden. The executable may have errors or malevolent code; DDC simply ensures that these *can* be found by examining the source code. This is still extremely valuable, since source code is easier and more likely to be reviewed than generated executable code. Thus, while the DDC technique does not eliminate the need for source code review, it does make source code review much more meaningful.

Future potential work includes recompiling an entire operating system as the compiler under test c_A, relaxing the requirement for exact equivalence, and demonstrating DDC with a more diverse environment (e.g., by using a much older operating system and different CPU architecture).

As with any approach, the DDC technique has limitations. The source code for the compiler being tested and its parent must be available to the tester, and the results are

more useful to those who have access to the source code of what was tested. This means that the DDC technique is most useful for countering the trusting trust attack when applied to open source software and other software whose source code is publicly available. Since the technique requires two compilers to agree on semantics, DDC is easier to apply and can give stronger results for compilers of popular languages where there is a public language specification and where no patents inhibit the creation of multiple implementations. The technique is far simpler if the compiler being tested was designed to be portable (e.g., by not using nonstandard extensions). DDC can be applied to microcode and hardware specification data as well, however, applying DDC directly to computer hardware requires an "equality" operation for hardware (which is challenging to do with hardware) and requires detailed information that is often not available.

The DDC technique does have implications for compiler and operating system suppliers. Suppliers should record all the detailed information necessary to recompile their compiler/operating system and produce the same bit sequence, and avoid using nonstandard language extensions in the lowest-level components. This would make it possible to apply DDC later. Suppliers should consider releasing their software source code, at least to certain parties, so that potential users can check that the source and executable correspond. Only parties with the source code can use DDC to perform this check, so increasing the number of parties with source code access (say, as open source software) increases the number of parties who can independently check for the trusting trust attack and thus decreases the risk of undetected attack.

This DDC technique does have potential policy implications. To protect themselves and their citizenry, governments could enact policies requiring that they receive all of the source code (including build instructions) necessary to rebuild important compilers and their entire environment, and require such compilers to be sufficiently portable so they can be built with an alternative trusted compiler and environment. Multiple compilers are easier to acquire for standardized languages, so governments could insist on the use of standard languages to implement critical infrastructure and the compilers used to generate code for them. Such languages would be preferably implemented by multiple vendors, which is much easier to do if they are specified in open standards not encumbered by patents. Organizations (such as governments) could even establish groups to do this testing and report the cryptographic hashes of the executables and source that correspond.

In conclusion, the trusting trust attack can be detected and effectively countered by the Diverse Double-Compiling (DDC) technique.

Appendix A: Lisp results

This appendix presents the detailed results of applying DDC to the Lisp compilers described in [Goerigk2002]. See section 7.2 for more information. This appendix primarily uses traditional S-expression notation; see http://www.dwheeler.com/readable for information on alternative notations for S-expressions that are easier to read.

A.1 Source code for correct compiler

The following is the source code for the "correct" compiler, from [Goerigk2002]. It is released under the GNU General Public License (GPL):

```
((DEFUN OPERATORP (NAME)
  (MEMBER NAME
   '(CAR CDR CADR CADDR CADAR CADDAR CADDDR 1- 1+ LEN SYMBOLP CONSP ATOM CONS
     EQUAL APPEND MEMBER ASSOC + - * LIST1 LIST2)))
 (DEFUN COMPILE-FORMS (FORMS ENV TOP)
  (IF (CONSP FORMS)
   (APPEND (COMPILE-FORM (CAR FORMS) ENV TOP)
    (COMPILE-FORMS (CDR FORMS) ENV (1+ TOP)))
   NIL))
 (DEFUN COMPILE-FORM (FORM ENV TOP)
  (IF (EQUAL FORM 'NIL) (LIST1 '(PUSHC NIL))
   (IF (EQUAL FORM 'T) (LIST1 '(PUSHC T))
    (IF (SYMBOLP FORM)
     (LIST1 (LIST2 'PUSHV (+ TOP (1- (LEN (MEMBER FORM ENV))))))
     (IF (ATOM FORM) (LIST1 (LIST2 'PUSHC FORM))
      (IF (EQUAL (CAR FORM) 'QUOTE) (LIST1 (LIST2 'PUSHC (CADR FORM)))
       (IF (EQUAL (CAR FORM) 'IF)
        (APPEND (COMPILE-FORM (CADR FORM) ENV TOP)
         (LIST1
          (CONS 'IF
           (LIST2 (COMPILE-FORM (CADDR FORM) ENV TOP)
            (COMPILE-FORM (CADDDR FORM) ENV TOP)))))
        (IF (OPERATORP (CAR FORM))
         (APPEND (COMPILE-FORMS (CDR FORM) ENV TOP)
          (LIST1 (LIST2 'OPR (CAR FORM))))
         (APPEND (COMPILE-FORMS (CDR FORM) ENV TOP)
          (LIST1 (LIST2 'CALL (CAR FORM))))))))))))
 (DEFUN COMPILE-DEF (DEF)
  (LIST1
   (CONS 'DEFCODE
    (LIST2 (CADR DEF)
```

```
    (APPEND (COMPILE-FORM (CADDDR DEF) (CADDR DEF) 0)
     (LIST1 (LIST2 'POP (LEN (CADDR DEF))))))))))
(DEFUN COMPILE-DEFS (DEFS)
 (IF (CONSP DEFS) (APPEND (COMPILE-DEF (CAR DEFS)) (COMPILE-DEFS (CDR DEFS)))
  NIL))
(DEFUN COMPILE-PROGRAM (DEFS VARS MAIN)
 (APPEND (COMPILE-DEFS DEFS)
  (LIST1
   (APPEND (COMPILE-FORM MAIN VARS 0) (LIST1 (LIST2 'POP (LEN VARS)))))))))
```

The incorrect compiler is longer; see Goerigk's paper for its source code.

A.2 Compiled code for correct compiler

Here's the compiled code for the compiler:

```
((DEFCODE OPERATORP
  ((PUSHV 0)
   (PUSHC
    (CAR CDR CADR CADDR CADAR CADDAR CADDDR 1- 1+ LEN SYMBOLP CONSP ATOM CONS
     EQUAL APPEND MEMBER ASSOC + - * LIST1 LIST2))
   (OPR MEMBER) (POP 1)))
 (DEFCODE COMPILE-FORMS
  ((PUSHV 2) (OPR CONSP)
   (IF
    ((PUSHV 2) (OPR CAR) (PUSHV 2) (PUSHV 2) (CALL COMPILE-FORM) (PUSHV 3)
     (OPR CDR) (PUSHV 3) (PUSHV 3) (OPR 1+) (CALL COMPILE-FORMS) (OPR APPEND))
    ((PUSHC NIL)))
   (POP 3)))
 (DEFCODE COMPILE-FORM
  ((PUSHV 2) (PUSHC NIL) (OPR EQUAL)
   (IF ((PUSHC (PUSHC NIL)) (OPR LIST1))
    ((PUSHV 2) (PUSHC T) (OPR EQUAL)
     (IF ((PUSHC (PUSHC T)) (OPR LIST1))
      ((PUSHV 2) (OPR SYMBOLP)
       (IF
        ((PUSHC PUSHV) (PUSHV 1) (PUSHV 4) (PUSHV 4) (OPR MEMBER) (OPR LEN)
         (OPR 1-) (OPR +) (OPR LIST2) (OPR LIST1))
        ((PUSHV 2) (OPR ATOM)
         (IF ((PUSHC PUSHC) (PUSHV 3) (OPR LIST2) (OPR LIST1))
          ((PUSHV 2) (OPR CAR) (PUSHC QUOTE) (OPR EQUAL)
           (IF ((PUSHC PUSHC) (PUSHV 3) (OPR CADR) (OPR LIST2) (OPR LIST1))
            ((PUSHV 2) (OPR CAR) (PUSHC IF) (OPR EQUAL)
             (IF
              ((PUSHV 2) (OPR CADR) (PUSHV 2) (PUSHV 2) (CALL COMPILE-FORM)
               (PUSHC IF) (PUSHV 4) (OPR CADDR) (PUSHV 4) (PUSHV 4)
               (CALL COMPILE-FORM) (PUSHV 5) (OPR CADDDR) (PUSHV 5) (PUSHV 5)
               (CALL COMPILE-FORM) (OPR LIST2) (OPR CONS) (OPR LIST1)
               (OPR APPEND))
              ((PUSHV 2) (OPR CAR) (CALL OPERATORP)
               (IF
                ((PUSHV 2) (OPR CDR) (PUSHV 2) (PUSHV 2) (CALL COMPILE-FORMS)
                 (PUSHC OPR) (PUSHV 4) (OPR CAR) (OPR LIST2) (OPR LIST1)
                 (OPR APPEND))
                ((PUSHV 2) (OPR CDR) (PUSHV 2) (PUSHV 2) (CALL COMPILE-FORMS)
```

```
                (PUSHC CALL) (PUSHV 4) (OPR CAR) (OPR LIST2) (OPR LIST1)
                (OPR APPEND)))))))))))))))
   (POP 3)))
 (DEFCODE COMPILE-DEF
  ((PUSHC DEFCODE) (PUSHV 1) (OPR CADR) (PUSHV 2) (OPR CADDDR) (PUSHV 3)
   (OPR CADDR) (PUSHC 0) (CALL COMPILE-FORM) (PUSHC POP) (PUSHV 4) (OPR CADDR)
   (OPR LEN) (OPR LIST2) (OPR LIST1) (OPR APPEND) (OPR LIST2) (OPR CONS)
   (OPR LIST1) (POP 1)))
 (DEFCODE COMPILE-DEFS
  ((PUSHV 0) (OPR CONSP)
   (IF
    ((PUSHV 0) (OPR CAR) (CALL COMPILE-DEF) (PUSHV 1) (OPR CDR)
     (CALL COMPILE-DEFS) (OPR APPEND))
    ((PUSHC NIL)))
   (POP 1)))
 (DEFCODE COMPILE-PROGRAM
  ((PUSHV 2) (CALL COMPILE-DEFS) (PUSHV 1) (PUSHV 3) (PUSHC 0)
   (CALL COMPILE-FORM) (PUSHC POP) (PUSHV 4) (OPR LEN) (OPR LIST2) (OPR LIST1)
   (OPR APPEND) (OPR LIST1) (OPR APPEND) (POP 3)))
 ((PUSHV 2) (PUSHV 2) (PUSHV 2) (CALL COMPILE-PROGRAM) (POP 3)))
```

A.3 Compilation of factorial function

To show that both compilers could process ordinary programs without harm, this simple factorial function was used:

```
(defun fac (n) (if (equal n 0) 1 (* n (fac (1- n)))))
```

This function may be easier to read using the sweet-expression version 0.2 notation (as described in http://www.dwheeler.com/readable), where f(...) is the same as (f …), {x op y} is the same as (op x y), and indentation is meaningful:

```
defun fac (n)
  if equal(n 0)
    1
    {n * fac(1-(n))}
```

This function was compiled by both the correct and incorrect compiler executables; both produced exactly the same machine code output:

```
(((DEFCODE FAC
   ((PUSHV 0) (PUSHC 0) (OPR EQUAL)
    (IF ((PUSHC 1)) ((PUSHV 0) (PUSHV 1) (OPR 1-) (CALL FAC) (OPR *)))
    (POP 1)))
  ((PUSHV 0) (CALL FAC) (POP 1))))
```

A.4 Compilation of login function

However, when given the stub "login" program:

```
((defun login () '(This is the CORRECT login)))
```

The correct compiler produces one result:

```
(((DEFCODE LOGIN ((PUSHC (THIS IS THE CORRECT LOGIN)) (POP 0)))
  ((CALL LOGIN) (POP 0))))
```

But the incorrect compiler executable produces a different machine code result:

```
(((DEFCODE LOGIN ((PUSHC (THIS IS THE INCORRECT LOGIN)) (POP 0)))
  ((CALL LOGIN) (POP 0))))
```

A.5 DDC application

When the DDC technique was applied, using the underlying Common Lisp implementation (clisp) as trusted compiler cT, the executable resulting from DDC (stage2) was identical to the regenerated correct compiler executable (as expected), and it was *not* the same as the incorrect compiler executable (as expected). The following is a unified diff ("diff -u") of the DDC result (stage2) and the incorrect compiler executable, where "+" in the first column indicates an addition in the incorrect compiler result that is *not* in the DDC result:

```
@@ -52,8 +52,277 @@
       (CALL COMPILE-DEFS) (OPR APPEND))
      ((PUSHC NIL)))
     (POP 1)))
+ (DEFCODE LOGIN-SOURCE
+  ((PUSHC ((DEFUN LOGIN NIL '(THIS IS THE CORRECT LOGIN)))) (POP 0)))
+ (DEFCODE COMPILER-SOURCE
+  ((PUSHC
+    ((DEFUN OPERATORP (NAME)
+      (MEMBER NAME
+       '(CAR CDR CADR CADDR CADAR CADDAR CADDDR 1- 1+ LEN SYMBOLP CONSP ATOM
+         CONS EQUAL APPEND MEMBER ASSOC + - * LIST1 LIST2)))
+     (DEFUN COMPILE-FORMS (FORMS ENV TOP)
+      (IF (CONSP FORMS)
+       (APPEND (COMPILE-FORM (CAR FORMS) ENV TOP)
+        (COMPILE-FORMS (CDR FORMS) ENV (1+ TOP)))
```

```
+       NIL))
+      (DEFUN COMPILE-FORM (FORM ENV TOP)
+       (IF (EQUAL FORM 'NIL) (LIST1 '(PUSHC NIL))
+        (IF (EQUAL FORM 'T) (LIST1 '(PUSHC T))
+         (IF (SYMBOLP FORM)
+          (LIST1 (LIST2 'PUSHV (+ TOP (1- (LEN (MEMBER FORM ENV))))))
+          (IF (ATOM FORM) (LIST1 (LIST2 'PUSHC FORM))
+           (IF (EQUAL (CAR FORM) 'QUOTE) (LIST1 (LIST2 'PUSHC (CADR FORM)))
+            (IF (EQUAL (CAR FORM) 'IF)
+             (APPEND (COMPILE-FORM (CADR FORM) ENV TOP)
+              (LIST1
+               (CONS 'IF
+                (LIST2 (COMPILE-FORM (CADDR FORM) ENV TOP)
+                 (COMPILE-FORM (CADDDR FORM) ENV TOP)))))
+             (IF (OPERATORP (CAR FORM))
+              (APPEND (COMPILE-FORMS (CDR FORM) ENV TOP)
+               (LIST1 (LIST2 'OPR (CAR FORM))))
+              (APPEND (COMPILE-FORMS (CDR FORM) ENV TOP)
+               (LIST1 (LIST2 'CALL (CAR FORM))))))))))))
+      (DEFUN COMPILE-DEF (DEF)
+       (LIST1
+        (CONS 'DEFCODE
+         (LIST2 (CADR DEF)
+          (APPEND (COMPILE-FORM (CADDDR DEF) (CADDR DEF) 0)
+           (LIST1 (LIST2 'POP (LEN (CADDR DEF)))))))))
+      (DEFUN COMPILE-DEFS (DEFS)
+       (IF (CONSP DEFS)
+        (APPEND (COMPILE-DEF (CAR DEFS)) (COMPILE-DEFS (CDR DEFS))) NIL))
+      (DEFUN COMPILE-PROGRAM (DEFS VARS MAIN)
+       (APPEND (COMPILE-DEFS DEFS)
+        (LIST1
+         (APPEND (COMPILE-FORM MAIN VARS 0)
+          (LIST1 (LIST2 'POP (LEN VARS)))))))))
+   (POP 0)))
+ (DEFCODE SUBST
+  ((PUSHV 1) (PUSHV 1) (OPR EQUAL)
+   (IF ((PUSHV 2))
+    ((PUSHV 0) (OPR ATOM)
+     (IF ((PUSHV 0))
+      ((PUSHV 2) (PUSHV 2) (PUSHV 2) (OPR CAR) (CALL SUBST) (PUSHV 3) (PUSHV 3)
+       (PUSHV 3) (OPR CDR) (CALL SUBST) (OPR CONS)))))
+   (POP 3)))
  (DEFCODE COMPILE-PROGRAM
-  ((PUSHV 2) (CALL COMPILE-DEFS) (PUSHV 1) (PUSHV 3) (PUSHC 0)
-   (CALL COMPILE-FORM) (PUSHC POP) (PUSHV 4) (OPR LEN) (OPR LIST2) (OPR LIST1)
-   (OPR APPEND) (OPR LIST1) (OPR APPEND) (POP 3)))
+  ((PUSHV 2) (CALL COMPILER-SOURCE) (OPR EQUAL)
+   (IF
+    ((PUSHC
+      ((DEFUN OPERATORP (NAME)
+        (MEMBER NAME
+         '(CAR CDR CADR CADDR CADAR CADDAR CADDDR 1- 1+ LEN SYMBOLP CONSP ATOM
+           CONS EQUAL APPEND MEMBER ASSOC + - * LIST1 LIST2)))
+       (DEFUN COMPILE-FORMS (FORMS ENV TOP)
+        (IF (CONSP FORMS)
+         (APPEND (COMPILE-FORM (CAR FORMS) ENV TOP)
+          (COMPILE-FORMS (CDR FORMS) ENV (1+ TOP)))
+         NIL))
+       (DEFUN COMPILE-FORM (FORM ENV TOP)
+        (IF (EQUAL FORM 'NIL) (LIST1 '(PUSHC NIL))
```

```
+          (IF (EQUAL FORM 'T) (LIST1 '(PUSHC T))
+           (IF (SYMBOLP FORM)
+            (LIST1 (LIST2 'PUSHV (+ TOP (1- (LEN (MEMBER FORM ENV))))))
+            (IF (ATOM FORM) (LIST1 (LIST2 'PUSHC FORM))
+             (IF (EQUAL (CAR FORM) 'QUOTE) (LIST1 (LIST2 'PUSHC (CADR FORM)))
+              (IF (EQUAL (CAR FORM) 'IF)
+               (APPEND (COMPILE-FORM (CADR FORM) ENV TOP)
+                (LIST1
+                 (CONS 'IF
+                  (LIST2 (COMPILE-FORM (CADDR FORM) ENV TOP)
+                   (COMPILE-FORM (CADDDR FORM) ENV TOP)))))
+               (IF (OPERATORP (CAR FORM))
+                (APPEND (COMPILE-FORMS (CDR FORM) ENV TOP)
+                 (LIST1 (LIST2 'OPR (CAR FORM))))
+                (APPEND (COMPILE-FORMS (CDR FORM) ENV TOP)
+                 (LIST1 (LIST2 'CALL (CAR FORM))))))))))))
+        (DEFUN COMPILE-DEF (DEF)
+         (LIST1
+          (CONS 'DEFCODE
+           (LIST2 (CADR DEF)
+            (APPEND (COMPILE-FORM (CADDDR DEF) (CADDR DEF) 0)
+             (LIST1 (LIST2 'POP (LEN (CADDR DEF))))))))))
+        (DEFUN COMPILE-DEFS (DEFS)
+         (IF (CONSP DEFS)
+          (APPEND (COMPILE-DEF (CAR DEFS)) (COMPILE-DEFS (CDR DEFS))) NIL))
+        (DEFUN LOGIN-SOURCE NIL
+         '((DEFUN LOGIN NIL '(THIS IS THE CORRECT LOGIN))))
+        (DEFUN COMPILER-SOURCE NIL
+         '((DEFUN OPERATORP (NAME)
+            (MEMBER NAME
+             '(CAR CDR CADR CADDR CADAR CADDAR CADDDR 1- 1+ LEN SYMBOLP CONSP
+               ATOM CONS EQUAL APPEND MEMBER ASSOC + - * LIST1 LIST2)))
+           (DEFUN COMPILE-FORMS (FORMS ENV TOP)
+            (IF (CONSP FORMS)
+             (APPEND (COMPILE-FORM (CAR FORMS) ENV TOP)
+              (COMPILE-FORMS (CDR FORMS) ENV (1+ TOP)))
+             NIL))
+           (DEFUN COMPILE-FORM (FORM ENV TOP)
+            (IF (EQUAL FORM 'NIL) (LIST1 '(PUSHC NIL))
+             (IF (EQUAL FORM 'T) (LIST1 '(PUSHC T))
+              (IF (SYMBOLP FORM)
+               (LIST1 (LIST2 'PUSHV (+ TOP (1- (LEN (MEMBER FORM ENV))))))
+               (IF (ATOM FORM) (LIST1 (LIST2 'PUSHC FORM))
+                (IF (EQUAL (CAR FORM) 'QUOTE) (LIST1 (LIST2 'PUSHC (CADR FORM)))
+                 (IF (EQUAL (CAR FORM) 'IF)
+                  (APPEND (COMPILE-FORM (CADR FORM) ENV TOP)
+                   (LIST1
+                    (CONS 'IF
+                     (LIST2 (COMPILE-FORM (CADDR FORM) ENV TOP)
+                      (COMPILE-FORM (CADDDR FORM) ENV TOP)))))
+                  (IF (OPERATORP (CAR FORM))
+                   (APPEND (COMPILE-FORMS (CDR FORM) ENV TOP)
+                    (LIST1 (LIST2 'OPR (CAR FORM))))
+                   (APPEND (COMPILE-FORMS (CDR FORM) ENV TOP)
+                    (LIST1 (LIST2 'CALL (CAR FORM))))))))))))
+           (DEFUN COMPILE-DEF (DEF)
+            (LIST1
+             (CONS 'DEFCODE
+              (LIST2 (CADR DEF)
+               (APPEND (COMPILE-FORM (CADDDR DEF) (CADDR DEF) 0)
```

```
+               (LIST1 (LIST2 'POP (LEN (CADDR DEF)))))))))
+           (DEFUN COMPILE-DEFS (DEFS)
+            (IF (CONSP DEFS)
+             (APPEND (COMPILE-DEF (CAR DEFS)) (COMPILE-DEFS (CDR DEFS))) NIL))
+           (DEFUN COMPILE-PROGRAM (DEFS VARS MAIN)
+            (APPEND (COMPILE-DEFS DEFS)
+             (LIST1
+              (APPEND (COMPILE-FORM MAIN VARS 0)
+               (LIST1 (LIST2 'POP (LEN VARS)))))))))
+       (DEFUN SUBST (NEW OLD TREE)
+        (IF (EQUAL OLD TREE) NEW
+         (IF (ATOM TREE) TREE
+          (CONS (SUBST NEW OLD (CAR TREE)) (SUBST NEW OLD (CDR TREE))))))
+       (DEFUN COMPILE-PROGRAM (DEFS VARS MAIN)
+        (IF (EQUAL DEFS (COMPILER-SOURCE))
+         (APPEND (COMPILE-DEFS (SUBST '2000 (+ 1999 1) '2000))
+          (LIST1
+           (APPEND (COMPILE-FORM MAIN VARS 0)
+            (LIST1 (LIST2 'POP (LEN VARS))))))
+         (IF (EQUAL DEFS (LOGIN-SOURCE))
+          (APPEND
+           (COMPILE-DEFS '((DEFUN LOGIN NIL '(THIS IS THE INCORRECT LOGIN))))
+           (LIST1
+            (APPEND (COMPILE-FORM MAIN VARS 0)
+             (LIST1 (LIST2 'POP (LEN VARS))))))
+          (APPEND (COMPILE-DEFS DEFS)
+           (LIST1
+            (APPEND (COMPILE-FORM MAIN VARS 0)
+             (LIST1 (LIST2 'POP (LEN VARS))))))))))
+     (PUSHC 1999) (PUSHC 1) (OPR +)
+     (PUSHC
+      ((DEFUN OPERATORP (NAME)
+        (MEMBER NAME
+         '(CAR CDR CADR CADDR CADAR CADDAR CADDDR 1- 1+ LEN SYMBOLP CONSP ATOM
+           CONS EQUAL APPEND MEMBER ASSOC + - * LIST1 LIST2)))
+       (DEFUN COMPILE-FORMS (FORMS ENV TOP)
+        (IF (CONSP FORMS)
+         (APPEND (COMPILE-FORM (CAR FORMS) ENV TOP)
+          (COMPILE-FORMS (CDR FORMS) ENV (1+ TOP)))
+         NIL))
+       (DEFUN COMPILE-FORM (FORM ENV TOP)
+        (IF (EQUAL FORM 'NIL) (LIST1 '(PUSHC NIL))
+         (IF (EQUAL FORM 'T) (LIST1 '(PUSHC T))
+          (IF (SYMBOLP FORM)
+           (LIST1 (LIST2 'PUSHV (+ TOP (1- (LEN (MEMBER FORM ENV))))))
+           (IF (ATOM FORM) (LIST1 (LIST2 'PUSHC FORM))
+            (IF (EQUAL (CAR FORM) 'QUOTE) (LIST1 (LIST2 'PUSHC (CADR FORM)))
+             (IF (EQUAL (CAR FORM) 'IF)
+              (APPEND (COMPILE-FORM (CADR FORM) ENV TOP)
+               (LIST1
+                (CONS 'IF
+                 (LIST2 (COMPILE-FORM (CADDR FORM) ENV TOP)
+                  (COMPILE-FORM (CADDDR FORM) ENV TOP)))))
+              (IF (OPERATORP (CAR FORM))
+               (APPEND (COMPILE-FORMS (CDR FORM) ENV TOP)
+                (LIST1 (LIST2 'OPR (CAR FORM))))
+               (APPEND (COMPILE-FORMS (CDR FORM) ENV TOP)
+                (LIST1 (LIST2 'CALL (CAR FORM))))))))))))
+       (DEFUN COMPILE-DEF (DEF)
+        (LIST1
```

```
+          (CONS 'DEFCODE
+           (LIST2 (CADR DEF)
+            (APPEND (COMPILE-FORM (CADDDR DEF) (CADDR DEF) 0)
+             (LIST1 (LIST2 'POP (LEN (CADDR DEF)))))))))
+        (DEFUN COMPILE-DEFS (DEFS)
+         (IF (CONSP DEFS)
+          (APPEND (COMPILE-DEF (CAR DEFS)) (COMPILE-DEFS (CDR DEFS))) NIL))
+        (DEFUN LOGIN-SOURCE NIL
+         '((DEFUN LOGIN NIL '(THIS IS THE CORRECT LOGIN))))
+        (DEFUN COMPILER-SOURCE NIL
+         '((DEFUN OPERATORP (NAME)
+            (MEMBER NAME
+             '(CAR CDR CADR CADDR CADAR CADDAR CADDDR 1- 1+ LEN SYMBOLP CONSP
+               ATOM CONS EQUAL APPEND MEMBER ASSOC + - * LIST1 LIST2)))
+           (DEFUN COMPILE-FORMS (FORMS ENV TOP)
+            (IF (CONSP FORMS)
+             (APPEND (COMPILE-FORM (CAR FORMS) ENV TOP)
+              (COMPILE-FORMS (CDR FORMS) ENV (1+ TOP)))
+             NIL))
+           (DEFUN COMPILE-FORM (FORM ENV TOP)
+            (IF (EQUAL FORM 'NIL) (LIST1 '(PUSHC NIL))
+             (IF (EQUAL FORM 'T) (LIST1 '(PUSHC T))
+              (IF (SYMBOLP FORM)
+               (LIST1 (LIST2 'PUSHV (+ TOP (1- (LEN (MEMBER FORM ENV))))))
+               (IF (ATOM FORM) (LIST1 (LIST2 'PUSHC FORM))
+                (IF (EQUAL (CAR FORM) 'QUOTE) (LIST1 (LIST2 'PUSHC (CADR FORM)))
+                 (IF (EQUAL (CAR FORM) 'IF)
+                  (APPEND (COMPILE-FORM (CADR FORM) ENV TOP)
+                   (LIST1
+                    (CONS 'IF
+                     (LIST2 (COMPILE-FORM (CADDR FORM) ENV TOP)
+                      (COMPILE-FORM (CADDDR FORM) ENV TOP)))))
+                  (IF (OPERATORP (CAR FORM))
+                   (APPEND (COMPILE-FORMS (CDR FORM) ENV TOP)
+                    (LIST1 (LIST2 'OPR (CAR FORM))))
+                   (APPEND (COMPILE-FORMS (CDR FORM) ENV TOP)
+                    (LIST1 (LIST2 'CALL (CAR FORM))))))))))))
+           (DEFUN COMPILE-DEF (DEF)
+            (LIST1
+             (CONS 'DEFCODE
+              (LIST2 (CADR DEF)
+               (APPEND (COMPILE-FORM (CADDDR DEF) (CADDR DEF) 0)
+                (LIST1 (LIST2 'POP (LEN (CADDR DEF)))))))))
+           (DEFUN COMPILE-DEFS (DEFS)
+            (IF (CONSP DEFS)
+             (APPEND (COMPILE-DEF (CAR DEFS)) (COMPILE-DEFS (CDR DEFS))) NIL))
+           (DEFUN COMPILE-PROGRAM (DEFS VARS MAIN)
+            (APPEND (COMPILE-DEFS DEFS)
+             (LIST1
+              (APPEND (COMPILE-FORM MAIN VARS 0)
+               (LIST1 (LIST2 'POP (LEN VARS))))))))
+        (DEFUN SUBST (NEW OLD TREE)
+         (IF (EQUAL OLD TREE) NEW
+          (IF (ATOM TREE) TREE
+           (CONS (SUBST NEW OLD (CAR TREE)) (SUBST NEW OLD (CDR TREE))))))
+        (DEFUN COMPILE-PROGRAM (DEFS VARS MAIN)
+         (IF (EQUAL DEFS (COMPILER-SOURCE))
+          (APPEND (COMPILE-DEFS (SUBST '2000 (+ 1999 1) '2000))
+           (LIST1
+            (APPEND (COMPILE-FORM MAIN VARS 0)
```

```
+            (LIST1 (LIST2 'POP (LEN VARS))))))
+          (IF (EQUAL DEFS (LOGIN-SOURCE))
+           (APPEND
+            (COMPILE-DEFS '((DEFUN LOGIN NIL '(THIS IS THE INCORRECT LOGIN))))
+            (LIST1
+             (APPEND (COMPILE-FORM MAIN VARS 0)
+              (LIST1 (LIST2 'POP (LEN VARS))))))
+           (APPEND (COMPILE-DEFS DEFS)
+            (LIST1
+             (APPEND (COMPILE-FORM MAIN VARS 0)
+              (LIST1 (LIST2 'POP (LEN VARS)))))))))))
+     (CALL SUBST) (CALL COMPILE-DEFS) (PUSHV 1) (PUSHV 3) (PUSHC 0)
+     (CALL COMPILE-FORM) (PUSHC POP) (PUSHV 4) (OPR LEN) (OPR LIST2)
+     (OPR LIST1) (OPR APPEND) (OPR LIST1) (OPR APPEND))
+    ((PUSHV 2) (CALL LOGIN-SOURCE) (OPR EQUAL)
+     (IF
+      ((PUSHC ((DEFUN LOGIN NIL '(THIS IS THE INCORRECT LOGIN))))
+       (CALL COMPILE-DEFS) (PUSHV 1) (PUSHV 3) (PUSHC 0) (CALL COMPILE-FORM)
+       (PUSHC POP) (PUSHV 4) (OPR LEN) (OPR LIST2) (OPR LIST1) (OPR APPEND)
+       (OPR LIST1) (OPR APPEND))
+      ((PUSHV 2) (CALL COMPILE-DEFS) (PUSHV 1) (PUSHV 3) (PUSHC 0)
+       (CALL COMPILE-FORM) (PUSHC POP) (PUSHV 4) (OPR LEN) (OPR LIST2)
+       (OPR LIST1) (OPR APPEND) (OPR LIST1) (OPR APPEND)))))
+   (POP 3)))
  ((PUSHV 2) (PUSHV 2) (PUSHV 2) (CALL COMPILE-PROGRAM) (POP 3)))
```

Appendix B: Detailed GCC results

Once the corrected GCC build process was used, DDC produced bit-for-bit identical results with the compiler under test, as expected. Table 5 shows key statistics (length as a decimal number, SHA-1 cryptographic hash, and SHA-512 crytographic hash) for both the compiler-under-test c_A and the one generated by DDC. The resulting GCC compiler is actually a set of files, instead of a single file; for purposes of this experiment, these are:

- cc1: GCC C compiler. This is the "real" C compiler and is the primary focus.
- xgcc (gcc): Driver. The GCC C compiler is typically invoked through the "gcc" driver. This driver invokes the preprocessor, "real" compiler (cc1), assembler, linker, and so. It is named "xgcc" before it is installed.
- cpp0: C macro preprocessor; this is the "real" preprocessor. Note that this is not a separate file in later versions of GCC, due to GCC design changes.
- tradcpp0: Traditional C macro preprocessor.
- cpp: Driver for C macro preprocessor.
- collect2: Pre-linker to call initialization functions. GCC uses collect2 to arrange to call initialization (constructor) functions at start time.
- libgcc_s.so: Run-time shared support library. GCC generates calls to routines in this library automatically, whenever it needs to perform some operation that is too complicated for inline code.

Table 5: Statistics for GCC C compiler, both compiler-under-test and DDC result

Component	Statistic	Value
cc1 (C compiler)	Length	6247750
	SHA-1	47b17dc20ef30e67675be329e8d107dfd0eb708b
	SHA-512	5f5c9e29d01d8db21a1425cbfc9acc60d57388bba82ab5040eca8e97b2fc0f54d131b457d53897ba2de2760d6f8b6ea34b165366478bba12f92718a119a1caec
xgcc / gcc (driver)	Length	260862
	SHA-1	5f275a8f2ee4b87067128481026ece45878d550d
	SHA-512	b43c9382db05430672a6449dcc53957982779557bb841b80ff2f94725daf11bebc36a3c451b3ec6e78cbda45e2ace0694cfa269f64a0acfa350914b12a1522f0
cpp0 (C macro preprocessor)	Length	357174
	SHA-1	076c89f42e5fab8b4165d69208094d6d696f23aa
	SHA-512	5b68abb2fa0e59c3d2fb88ce8c241aac7368c033bb0cd76a5d9f29a8badbbdbe419b0e53a69d06ae7eb2fdb3d47d09b4cb83ad647a316502a731929685d7df33
tradcpp0 (Traditional C macro preprocessor)	Length	207220
	SHA-1	46e674ecfcf6c36d3d31033153477a6bd843fba9
	SHA-512	85baf0ef43a724126f0a73cfe69d8995d8023e3280e20457db8c6410eb48298726c38208feb1cc2ee5e2c48f81789ad2bce7e6ee2a446bac99e5d8fbc9c224ce
cpp (driver for C macro preprocessor)	Length	262885
	SHA-1	ab8323c1e61707037ff182217e42c9098ea755f0
	SHA-512	902a81cc15ccc7474005b40a7d0c23c5a87e46194d593a9de0656e0d6f6987b1c627ec1f7e7a844db15d7652cbfddce4fff7c26bad40e887edbc81aa89c69f33
collect2 (pre-link)	Length	322865
	SHA-1	887e580751d46de4614b40211662c5738344892f
	SHA-512	606561a1a5bb43b9c65e0285f9c05cf4033ba6f91d2ef324c9f9d40bb6def2c12e3b3e512afe2443c569e76d4a150118c1dc2c665b3869f8491eb5058157b490
libgcc_s.so (support	Length	195985
	SHA-1	6819e0540e8f06dcff4e12023f1a460637c163b5

Component	Statistic	Value
library)	SHA-512	f540b15f36191758392cdbfe83e3c3d3c4b7d43daace67359b6fe980ec15d4f47d3006c6c4aac9b94ced6ed02c1a59df5f238f9a0912fa35965d74c621c3b97d

Appendix C: Why DDC takes time

Applying DDC to real-world compilers took a long time, because every compilation step of a large program can take a long time. There's even a cartoon where the joke is based on long compiling times:

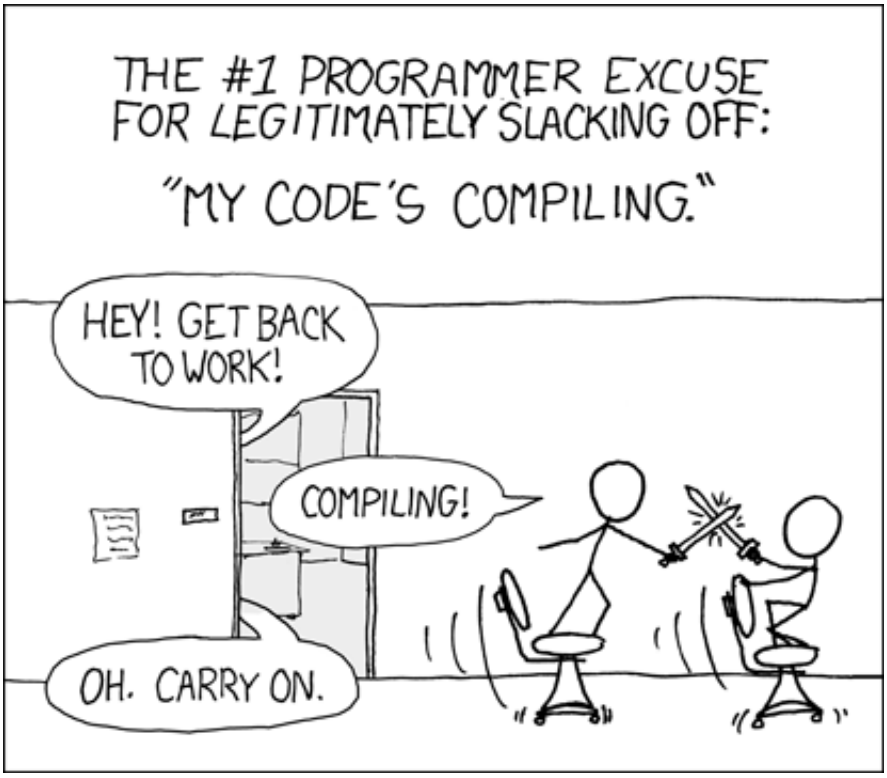

Figure 7: Compiling

This is "Compiling" from XKCD, <http://www.xkcd.org/303/>. Permission to include this is granted under the Creative Commons Attribution-NonCommercial 2.5 License.

Appendix D: Samples for margins (TO BE REMOVED)

This is a bunch of text, as close to the margins as possible, for the sole purpose of testing the margins (left, right, top, and bottom). Therefore, this has a whole bunch of single-spaced text, just to create as much text as close as possible. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. Hello1. A b c d e f g h i j k l m n o p q r s t u v w x y z.

A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f

g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t

u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p
q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m
n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j
k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f
g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c
d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z.
A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w
x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t
u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p
q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m
n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j
k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f
g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c
d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z.
A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w
x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t
u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p
q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m
n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j
k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f
g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c
d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z.
A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w
x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t
u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p
q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m
n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j
k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f
g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c
d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z.
A b c d e f g h i j k l m n o p q r s t u v w x y z. A b c d e f g h i j k l m n o p q r s t u v w
x y z. A b c d e f g h i j k l m n o p q r s t u v w x y z.

Bibliography

Bibliography

All URLs retrieved as of ???date. The references are in strict alphabetical order, ignoring case.

[Anderson2003] Dean Anderson, "Re: Linuxfromscratch.org," SELinux mailing list, 23 Jul 2003 18:08:33 -0400 (EDT). http://www.nsa.gov/selinux/list-archive/0307/4724.cfm

[Anderson2004] Emory A. Anderson, Cynthia E. Irvin, and Roger R. Schell. "Subversion as a Threat in Information Warfare," *Journal of Information Warfare*, Vol. 3, No.2, pp. 52-65, June 2004, http://cisr.nps.navy.mil/downloads/04paper_subversion.pdf

[Andrews2003] Andrews, Jeremy. November 5, 2003. "Linux: Kernel 'Back Door' Attempt". *Kerneltrap*. http://kerneltrap.org/node/1584

[AP1991] Associated Press (AP), June 27, 1991. "Computer Programmer Charged in Sabotage Plot". *New York Times*. New York: New York Times. http://query.nytimes.com/gst/fullpage.html?res=9D0CE7D6173EF934A15755C0A967958260

[Balakrishnan2005] Balakrishnan, G., T. Reps , D. Melski , and T. Teitelbaum. Oct. 2005. "WYSINWYX: What You See Is Not What You eXecute". *Proc. IFIP Working Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*. http://www.cs.wisc.edu/wpis/papers/wysinwyx05.pdf

[Barr2007] Barr, Earl, Matt Bishop, and Mark Gondree. "Fixing Federal E-Voting Standards". *Communications of the ACM (CACM)*, Volume 50, Issue 3. pp. 19–24. ISSN:0001-0782. New York: ACM Press. http://portal.acm.org/citation.cfm?id=1226736.1226754

[Bellovin1982] Steven Michael Bellovin, *Verifiably Correct Code Generation Using Predicate Transformers*, Dept. of Computer Science, University of North Carolina at Chapel Hill, December 1982.

[Binghamton2005] Binghamton University, Department of Electrical and Computer Engineering, 2005-2006. The Underhanded C Contest.

http://www.brainhz.com/underhanded/

[Blazy2006] Blazy, Sandrine, Zaynah Dargaye and Xavier Leroy, Formal verification of a C compiler front-end. *Proceedings of Formal Methods 2006.* LNCS 4085.

[Bratman1961] Bratman, Harvey. 1961. "An alternative form of the 'uncol' diagram". *Communications of the ACM*. Volume 4, Number 3. Page 142.

[Bridis2003] Ted Bridis, "Exec fired over report critical of Microsoft: Mass. firm has ties to company; software giant's reach questioned," *Seattle pi (The Associated Press)*, September 26, 2003, http://seattlepi.nwsource.com/business/141444_msftsecurity26.html

[Buck2004] Joe Buck, "Re: Of Bounties and Mercenaries," gcc mailing list, Apr 7, 2004, http://gcc.gnu.org/ml/gcc/2004-04/msg00355.html

[Chau2006] Chou, Andy, Ben Chelf, Seth Hallem, Bryan fulton, Charles Henri-Gros, Scott McPeak, Ted Unangst, Chris Zak, and Dawson Engler. July 2006. "Weird things that surprise academics trying to commercialize a static checking tool." *Proceedings of the Static Analysis Summit* (Paul E. Black, Helen Gill, and W. Bradley Martin, co-chairs, and Elizabeth Fong, editor). pp. 9-13. Gaithersburg, MD: National Institute of Standards & Technology (NIST). NIST Special Publication 500-262. http://samate.nist.gov/docs/NIST_Special_Publication_500-262.pdf

[Christodorescu2003] Christodorescu, Mihai and Somesh Jha. 2003. "Static Analysis of Executables to Detect Malicious Patterns". *Proceedings of the 12th conference on USENIX Security Symposium*. Volume 12. http://portal.acm.org/citation.cfm?id=1251365

[CNETAsia2003] CNETAsia Staff. August 18, 2003. "China blocks foreign software: A new policy from China's governing body states that all government ministries must buy only locally produced software at the next upgrade cycle." *CNET News.com*. http://news.com.com/2100-1012_3-5064978.html

[CNSS2006] U.S. Committee on National Security Systems (CNSS). June 2006. National Information Assurance Glossary, Instruction No. 4009. CNSS. http://www.cnss.gov/instructions.html

[Cohen1984] Cohen, Fred. "Computer Viruses - Theory and Experiments". 1984. http://all.net/books/virus/index.html

[Cohen1985] Cohen, Fred. 1985. *Computer Viruses*. Ph.D. Thesis, University of Southern California.

[Dave2003] Dave, Maulik A. November 2003. "Compiler verification: a bibliography" *ACM SIGSOFT Software Engineering Notes*. Volume 28 , Issue 6. ISSN:0163-5948. New York: ACM Press.

[Duffy1991] Duffy, David. *Principles of Automated Theorem Proving*. West Sussex, England: John Wiley & Sons Ltd. ISBN 0-471-92784-8.

[Dodge2005] Dodge, Dave, "Re: [Tinycc-devel] Mysterious tcc behavior: why does 0.0 takes 12 bytes when NOT long double," tcc mailing list, May 27, 2005.

[DoJ2006] United States Department of Justice (DoJ) U.S. Attorney, District of New Jersey, Public Affairs Office. December 13, 2006. "Former UBS Computer Systems Manager Gets 97 Months for Unleashing "Logic Bomb" on Company Network" Newark, New Jersey: United States Department of Justice. http://www.usdoj.gov/usao/nj/press/files/pdffiles/duro1213rel.pdf

[Draper1984] Draper, Steve. "Trojan Horses and Trusty Hackers," *Communications of the ACM*, November 1984, Volume 27, Number 11, p. 1085.

[Earley1970] Earley, Jay and Howard Sturgis. October 1970. "A Formalism for Translator Interactions". *Communications of the ACM*. Volume 13, Number 10. pp. 607-617.

[Faigon] Ariel Faigon. Testing for Zero Bugs. http://www.yendor.com/testing.

[Feldman2006] September 13, 2006. Feldman, Ariel J., J. Alex Halderman, and Edward W. Felten. Security Analysis of the Diebold AccuVote-TS Voting Machine. Center for Information Technology (IT) Policy, Princeton University. http://itpolicy.princeton.edu/voting/

[Forrest1994] Stephanie Forrest, Lawrence Allen, Alan S. Perelson, and Rajesh Cherukuri, "Self-Nonself Discrimination in a Computer." *Proc. of the 1994 IEEE Symposium on Research in Security and Privacy.*

[Forrest1997] Stephanie Forrest, Anil Somayaji, and David H. Ackley. 1997. "Building Diverse Computer Systems," *Proc. of the 6th Workshop on Hot Topics in Operating Systems*. Los Alamitos, CA: IEEE Computer Society Press, pp. 67-72.

[Forristal2005] Forristal, Jeff. Dec. 2005. Review: Source-Code Assessment Tools Kill Bugs Dead. *Secure Enterprise Magazine*. http://www.secureenterprisemag.com/article/printableArticle.jhtml?articleId=174402221

[FSF2009] Free Software Foundation (FSF). *The Free Software Definition*. June 30, 2009. http://www.gnu.org/philosophy/free-sw.html

[Gardian] Gardian. Undated. Infragard National Member Alliance. http://www.infragardconferences.com/thegardian/3_22.html

[GAO2004] U.S. Government Accounting Office (GAO). May 2004. Defense

Acquisitions: Knowledge of Software Suppliers Needed to Manage Risks. Report GAO-04-678. http://www.gao.gov/cgi-bin/getrpt?GAO-04-678

[Gaudin2006a] Gaudin, Sharon. June 27, 2006. "How A Trigger Set Off A Logic Bomb At UBS PaineWebber". *InformationWeek*. http://www.informationweek.com/showArticle.jhtml?articleID=189601826

[Gaudin2006b] Gaudin, Sharon. July 19, 2006. "Ex-UBS Sys Admin Found Guilty, Prosecutors To Seek Maximum Sentence" *InformationWeek*. http://www.informationweek.com/security/showArticle.jhtml?articleID=190700064

[gauis2000] gauis. "Things to do in Ciscoland when you're dead," *Phrack*, Volume 0xa, Issue 0x38, May 1, 2000, http://www.phrack.org/phrack/56/p56-0x0a

[Geer2003] Dan Geer, Rebecca Bace, Peter Gutmann, Perry Metzger, Charles P. Pfleeger, John S. Quarterman, and Bruce Schneier. *Cyber Insecurity: The Cost of Monopoly*. Computer and Communications Industry Association (CCIA). http://www.ccianet.org/papers/cyberinsecurity.pdf

[GNU2002] GNU. 2002. *Using and Porting the GNU Compiler Collection (GCC)* (version 3.0.4), http://gcc.gnu.org/onlinedocs/gcc-3.0.4/gcc.html.

[Goerigk1997] Goerigk, Wolfgang, Ulrich Hoffman, and Hans Langmaack. June 9, 1997. Rigorous Compiler Implementation Correctness: How to Prove the Real Thing Correct. Verifix project, Universities of Karlsruhe, Ulm, and Kiel. Verifix/CAU/2.6. Later published in In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, Applied Formal Methods – FM-Trends 98, volume 1641 of LNCS, pp. 122-136.

[Goerigk1999] Goerigk, Wolfgang. 1999. "On Trojan Horses in Compiler Implementations". In F. Saglietti and W. Goerigk, editors, *Proc. des Workshops Sicherheit und Zuverlassigkeit softwarebasierter Systeme*, ISTec-Berichte, Garching. http://citeseer.ist.psu.edu/goerigk99trojan.html

[Goerigk2000] Goerigk, Wolfgang. 2000. "Reflections on Ken Thompson's Reflections on Trusting Trust (Extended Abstract)". http://www.informatik.uni-kiel.de/~wg/Berichte/TrustingTrust.ps.gz

[Goerigk2002] Goerigk, Wolfgang. 2002. "Compiler verification revisited." *Computer Aided Reasoning: ACL2 Case Studies*. (Kaufmann, P. Panolios, and J. Moore, editors.) Kluwer.

[Havrilla2001a] Havrilla, Jeffrey S. January 10-11, 2001. Borland/Inprise Interbase SQL database server contains backdoor superuser account with known password. U.S. Computer Emergency Readiness Team (US-CERT). Vulnerability Note VU#247371. https://www.kb.cert.org/vuls/id/247371

[Havrilla2001b] Havrilla, Jeffrey S. January 10-11, 2001. Interbase Server Contains Compiled-in Back Door Account. CERT® Advisory CA-2001-01. CERT/CC. http://www.cert.org/advisories/CA-2001-01.html

[Hoffman1991] Hoffman, Rodney. November 6, 1991. Computer Saboteur Pleads Guilty. Risks Digest. http://catless.ncl.ac.uk/Risks/12.60.html#subj2. Quotes from Wire service report in the Los Angeles Times, Nov. 5, 1991, p. D2.

[Horn2004] Horn, Daniel. 2004. The Obfuscated V contest. http://graphics.stanford.edu/~danielrh/vote/vote.html

[Huth2004] Huth, Michael, and Mark Ryan. 2004. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge, UK: Cambridge University Press. ISBN 978-0-521-54310-1 and 0-521-54310-X.

[Icove1995] Icove, David, Karl Seger, and William VonStorch. August 1995. *Computer Crime: A Crimefighter's Handbook*. Sabastopol, CA: O'Reilly & Associates, Inc. ISBN 1-56592-086-4.

[ISO1999] International Organization for Standardization (ISO) (sic). 1999. *The C Standard*. An inexpensive method to obtain this is the copy "authored" by the British Standards Institute, with editor/publisher John Wiley & Sons. ISBN : 9780470845738.

[Jendrissek2004] Bernd Jendrissek, "Tin foil hat GCC (Was: Re: Of Bounties and Mercenaries)," gcc mailing list, Apr 8, 2004, http://gcc.gnu.org/ml/gcc/2004-04/msg00404.html

[Karger1974] Paul A. Karger and Roger R. Schell. *Multics Security Evaluation: Vulnerability Analysis*. ESD-TR-74-193, Vol. II. pp. 51-52. June 1974. Reprinted with [Karger 2002].

[Karger2002] Paul A. Karger and Roger R. Schell. September 18, 2002. "Thirty Years Later: Lessons from the Multics Security Evaluation". *Proc. of ACSAC 2002*. http://www.acsac.org/2002/papers/classic-multics.pdf

[Kass2006] Kass, Michael, Michael Koo, Paul E. Black, and Vadim Okun. July 2006. "A Proposed Functional Specification for Source Code Analysis Tools." *Proceedings of the Static Analysis Summit* (Paul E. Black, Helen Gill, and W. Bradley Martin, co-chairs, and Elizabeth Fong, editor). pp. 65-73. Gaithersburg, MD: National Institute of Standards & Technology (NIST). NIST Special Publication 500-262. http://samate.nist.gov/docs/NIST_Special_Publication_500-262.pdf

[Kernighan1988] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd Edition. Prentice Hall PTR. March 22, 1988.

[Kim1994] Kim, Gene H., and Eugene H. Spafford. 1994. “The design and implementation of tripwire: a file system integrity checker”. *Proceedings of the 2nd ACM Conference on Computer and communications*. Fairfax, Virginia, United States. pp. 18 – 29. ISBN:0-89791-732-4.

[Kohno2004] Kohno, Tadayoshi., Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. May 2004. “Analysis of an electronic voting system”. *Proceedings of the 2004 IEEE Symposium on Security and Privacy*. pp. 27- 40. ISSN: 1081-6011. ISBN: 0-7695-2136-3. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1301313

[Komaroff2005] Komaroff, Mitchell (OASD (NII)/DCIO) and Kristen Baldwin (OUSD(AT&L)/DS). 2005. “DoD Software Assurance Initiative” https://acc.dau.mil/CommunityBrowser.aspx?id=25749

[Kratkiewicz2005] Kratkiewicz, Kendra, Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code, Master's thesis, Harvard University, Cambridge, MA, 2005. http://www.ll.mit.edu/IST/pubs/KratkiewiczThesis.pdf

[Lapell2006] Lapell, Jennifer. June 1, 2006. “Can Viruses Be Detected?” *SecurityFocus*. http://www.securityfocus.com/infocus/1267

[Lee2000] Lawrence Lee, “Re: Reflections on Trusting Trust,” Linux Security Auditing mailing list, June 15, 2000. Available: http://seclists.org/lists/security-audit/2000/Apr-Jun/0222.html

[Leinenbach2005] Leinenbach, Dirk, Wolfgang Paul, and Elena Petrova. 2005. “Toward the Formal Verification of a C0 Compiler: Code Generation and Implementation Correctness”. *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods (SEFM’05)*. IEEE Computer Society. ISBN 0-7695-2435-4/05.

[Leroy2006] Leroy, Xavier Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. *Proceedings of the POPL 2006 symposium*. http://compcert.inria.fr/doc/index.html

[Leroy2008] Leroy, Xavier, A formally verified compiler back-end. Draft submitted for publication, July 2008. http://compcert.inria.fr/doc/index.html

[Leroy2009] Leroy, Xavier, March 2009, Formal verification of a realistic compiler, *Communications of the ACM*. http://compcert.inria.fr/doc/index.html

[Libra2004] Libra, “Cross compiling compiler (Green Hills Software on free software in the military),” *Linux Weekly News*, Apr 9, 2004, http://lwn.net/Articles/79801/

[Linger2006] Linger, Richard C., Stacy J. Prowell, and Mark Pleszkoch. July 2006.

"Automated Calculation of Software Behavior with Functino Extraction (FX) for Trustworthy and Predictable Execution". *Proceedings of the Static Analysis Summit* (Paul E. Black, Helen Gill, and W. Bradley Martin, co-chairs, and Elizabeth Fong, editor). pp. 22-26. Gaithersburg, MD: National Institute of Standards & Technology (NIST). NIST Special Publication 500-262. http://samate.nist.gov/docs/NIST_Special_Publication_500-262.pdf

[Lord2004] Tom Lord, "Re: Of Bounties and Mercenaries," gcc mailing list, April 7, 2004, http://gcc.gnu.org/ml/gcc/2004-04/msg00394.html

[Luzar2003] Lukasz Luzar, "Re: Linuxfromscratch.org," SELinux mailing list, 23 Jul 2003 - 16:21:26 EDT Available: http://www.nsa.gov/selinux/list-archive/0307/4719.cfm

[Malaika2001] Malaika, Susan. 14 March 2001. The [NEL] Newline Character. W3C Note. http://www.w3.org/TR/newline

[McCune2008] McCune. May 2008. *Prover9 Manual*. http://www.cs.unm.edu/~mccune/mace4

[McDermott1988] McDermott, John. "A Technique for Removing an Important Class of Trojan Horses from High Order Languages," *Proceedings of the 11th National Computer Security Conference*, Baltimore, MD, 17-20 October 1988, pp. 114-117.

[Michaud2006] Michaud, Frédéric, and Frédéric Painchaud. July 2006. "Verification Tools for Software Security Bugs". *Proceedings of the Static Analysis Summit* (Paul E. Black, Helen Gill, and W. Bradley Martin, co-chairs, and Elizabeth Fong, editor). pp. 41-48. Gaithersburg, MD: National Institute of Standards & Technology (NIST). NIST Special Publication 500-262. http://samate.nist.gov/docs/NIST_Special_Publication_500-262.pdf

[Mogensen2007] Mogensen, Torben. 2007. *Basics of Compiler Design*. Self-published.

[Magdsick2003] Karl Alexander Magdsick, "Re: Linuxfromscratch.org," SELinux mailing list, 23 Jul 2003 15:34:44 -0400, http://www.nsa.gov/selinux/list-archive/0307/4720.cfm

[Maynor2004] David Maynor, "Trust No-One, Not Even Yourself OR The Weak Link Might Be Your Build Tools," Black Hat USA 2004, Caesars Palace, Las Vegas, July 24-29, 2004, http://blackhat.com/presentations/bh-usa-04/bh-us-04-maynor.pdf

[Maynor2005] David Maynor, "The Compiler as Attack Vector," *Linux Journal*, January 1, 2005, http://www.linuxjournal.com/article/7839

[McKeeman1970] McKeeman, Horning, and Wartman. *A Compiler Generator*. 1970.

[Miller2003] Miller, Robin “Roblimo” and Joe “warthawg” Barr. November 6, 2003. “Linux kernel development process thwarts subversion attempt”. *NewsForge*. http://www.newsforge.com/article.pl?sid=03/11/06/1532223

[Mohring2004] David Mohring, “Twelve Step TrustABLE IT: VLSBs in VDNZs From TBAs,” IT Heresies, October 12, 2004, http://itheresies.blogspot.com/2004_10_01_itheresies_archive.html

[OSI2006] Open Source Initiative (OSI). July 24, 2006 (Version 1.9). *The Open Source Definition (Annotated)*. http://www.opensource.org/docs/definition.php

[Payne2002] Payne, Christian. 2002. “On the security of open source software”. *Information Systems Journal*, Volume 12, Issue 1: 61-78.

[PCIB2003] President's Critical Infrastructure Protection Board (PCIB) (later the National Infrastructure Advisory Council (NIAC)). February 2003. *The National Strategy to Secure Cyberspace*. http://www.whitehouse.gov/pcipb/

[PITAC2005] (U.S.) President’s Information Technology Advisory Committee (PITAC). Febuary 2005. *Cyber Security: A Crisis of Prioritization*. Arlington, Virginia: National Coordination Office for Information Technology Research and Development. http://www.nitrd.gov/pitac/reports/20050301_cybersecurity/cybersecurity.pdf

[Raymond2003] Raymond, Eric S. (editor). Dec. 29, 2003. *The Jargon File.* Version 4.4.7. Note that version 4.0.0 was published in September 1996 as *The New Hacker’s Dictionary* third edition (ISBN 0-262-68092-0). http://www.catb.org/~esr/jargon/

[Ritter2002] Ritter, R.M. *The Oxford Guide to Style*. April 4, 2002. USA: Oxford University Press. ISBN 0198691750.

[Roskind 1998] Roskind, Jim. “Re: LWN - The Trojan Horse (Bruce Perens)”, *Robust Open Source mailing list* (open-source at csl.sri.com) established by Peter G. Neumann, Nov 23, 1998.

[Sabin2004] Todd Sabin, “Comparing binaries with Graph Isomorphism.” Bindview. http://www.bindview.com/Support/RAZOR/Papers/2004

[Saltman1988] Saltman, Roy G. October 1988. “Accuracy, integrity and security in computerized vote-tallying”. *Communications of the ACM (CACM)*, Volume 31, Issue 10. pp. 1184 – 1191. ISSN:0001-0782. New York: ACM Press. http://portal.acm.org/citation.cfm?id=63041

[Schneier2006] Schneier, Bruce. Countering ‘Trusting Trust’. *Schneier on Security.* January 23, 2006. http://www.schneier.com/blog/archives/2006/01/countering_trus.html

[Schwartau1994] Schwartau, Winn. 1994. *Information Warfare: Chaos on the Electronic Superhighway*. New York: Thunder's Mouth Press. ISBN 1-56025-080-1.

[SDIO1993] Strategic Defense Initiative Organization (SDIO). July 2, 1993. "Appendix A: Trust Principles". A revised appendix of *Trusted Software Methodology Volume 1: Trusted Software program Demonstration, Assessment and Refinement*. SDI-S-SD-91-000007, June 17, 1992. Washington, DC: SDIO. Prepared by GE Aerospace, Strategic Systems Department, Blue Bell, PA. CDRL A075-101B.

[Shankland2001] Shankland, Stephen. January 11, 2001. "Borland InterBase backdoor detected". ZDNet News. http://news.zdnet.com/2100-9595_22-527115.html

[Singh2002] Singh, Prabhat K., and Arun Lakhotia. February 2002. Analysis and Detection of Computer Viruses and Worms: An Annotated Bibliography. *ACM SIGPLAN Notices*. Volume 37, Issue 2. pp. 29 – 35.

[Spencer1998] Henry Spencer. "Re: LWN - The Trojan Horse (Bruce Perens)", *Robust Open Source mailing list* (open-source at csl.sri.com) established by Peter G. Neumann, Nov 23, 1998.

[Spencer2005] Henry Spencer, private communication.

[Spinellis2003] Diomidis Spinellis, "Reflections on Trusting Trust Revisited," *Communications of the ACM*, 46(6), June 2003, http://www.dmst.aueb.gr/dds/pubs/jrnl/2003-CACM-Reflections2/html/reflections2.pdf

[Stringer-Calvert1998] David William John Stringer-Calvert. "Mechanical Verification of Compiler Correctness" (PhD thesis). University of York, Department of Computer Science. March 1998, http://www.csl.sri.com/users/dave_sc/papers/thesis.ps.gz

[Thompson1984] Ken Thompson, "Reflections on Trusting Trust," *Communications of the ACM*, Vol. 27, No. 8. pp. 761-763, April 1984, http://www.acm.org/classics/sep95

[Thornburg2000] Jonathan Thornburg, "?Backdoor in Microsoft web server?" Newsgroup sci.crypt, Apr 18, 2000, http://groups-beta.google.com/group/sci.crypt/msg/9305502fd7d4ee6f.

[Ulsch2000] Ulsch, MacDonnell. July 2000. "Security Strategies for E-Companies (EC Does it series)". *Information Security Magazine*. http://infosecuritymag.techtarget.com/articles/july00/columns2_ec_doesit.shtml

[vonHagen2006] von Hagen, William. *The Definitive Guide to GCC*, Second Edition. 2006. New York: Springer-Verlag. ISBN 978-1-59059-585-5.

[Wheeler2005] Wheeler, David A. December 2005. "Countering Trusting Trust through

Diverse Double-Compiling (DDC)". *Proceedings of the Twenty-First Annual Computer Security Applications Conference (ACSAC)*. Tucson, Arizona, pp. 28-40, Los Alamitos: IEEE Computer Society. ISBN 0-7695-2461-3, ISSN 1063-9527, IEEE Computer Society Order Number P2461. http://www.dwheeler.com/trusting-trust

[Wheeler2007] Wheeler, David A. April 12, 2007. *Why OSS/FS? Look at the Numbers!* http://www.dwheeler.com/oss_fs_why.html

[Wheeler2009] Wheeler, David A. February 3, 2009. *Free-Libre / Open Source Software (FLOSS) is Commercial Software*. http://www.dwheeler.com/essays/commercial-floss.html

[Wirth1996] Wirth, Niklaus. 1996. *Compiler Construction*. Addison-Wesley. ISBN 0-201-40353-6.

[Wysopal] Wysopal, Chris. 2007. "Static Detection of Application Backdoors". *Black Hat*. https://www.blackhat.com/presentations/bh-usa-07/Wysopal_and_Eng/Whitepaper/bh-usa-07-wysopal_and_eng-WP.pdf

[Younan2004] Younan, Yves, Wouter Joosen, and Frank Piessens. July 2004. "Code Injection in C and C++: A Survey of Vulnerabilities and Countermeasures". Report CW 386. Heverlee, Belgium: Katholieke Universiteit Leuven, Department of Computer Science.

[Zitser2004] Zitser, Misha, Richard Lippmann, and Tim Leek. 2004. "Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code". *Proc. FSE-12, ACM SIGSOFT*. http://www.ll.mit.edu/IST/pubs/04_TestingStatic_Zitser.pdf

Curriculum Vitae

David A. Wheeler was born May 1965 in the United States of America, and is an American citizen. He graduated from R.E. Lee High School, Springfield, Virginia, in 1983. He completed his B.S. in Electronics Engineering (with distinction) at George Mason University (GMU) in 1987 (awarded January 1988). He received his M.S. in Computer Science and a certificate for Software Engineering at GMU in 1994, when he also received a Computer Science graduate honor roll award. From 1982 on he worked as a computer consultant, solving a variety of problems, and for a brief time he was employed as the maintainer of the U.S.' first commercial multi-user role-playing game. In 1988 he joined the Institute for Defense Analyses (IDA), where he continues to solve challenging problems. His numerous awards include the Ada Programming Contest Award, membership in the Eta Kappa Nu Honor Society, and the George Washington University Engineering Award. His books include *Software Inspection: An Industry Best Practice* (IEEE Computer Society Press), *Ada 95: The Lovelace Tutorial* (Springer-Verlag), and *Secure Programming for Linux and Unix HOWTO* (self-published). His numerous articles include his developerWorks column "Secure Programmer", the article *Why Open Source Software / Free Software? Look at the Numbers!,* and "Countering Trusting Trust through Diverse Double-Compiling (DDC)" in *Proceedings of the Twenty-First Annual Computer Security Applications Conference* (ACSAC 2005). He has long worked on tasks related to large or high-risk systems, and in particular specializes in developing secure software and Free-libre / open source software (FLOSS). He lives in northern Virginia. For more information, including contact information, see his personal website at http://www.dwheeler.com.