Countering Trusting Trust through Diverse Double-Compiling

Author (not stated here to permit blind review) – version May 27, 2005

***Abstract*—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 attack goes undetected, even complete analysis of a system's source code will not find the malicious code that is running, and methods for detecting this particular attack are not widely known. This paper describes a practical technique, termed diverse double-compiling, that detects this attack and some unintended compiler defects as well. Simply recompile the purported source code twice: once with a second (trusted) compiler, and again using the result of the first compilation; if the result is bit-for-bit identical with the untrusted binary, then the source code accurately represents the binary. This technique has been mentioned informally, but to the author's knowledge the issues and ramifications have not been publicly identified or discussed in detail, nor has a public demonstration been made. This paper describes the technique, justifies it, demonstrates it, and provides lessons learned.[1]**

***Index Terms*—Trusting trust, compilers, trusted compilers, Trojan horse, diversity, diverse double-compiling.**

I. Introduction

Many product security evaluations examine source code, under the assumption that the source code accurately represents the product being examined. Naïve developers presume that this can be assured simply by recompiling the source code to see if the same binary results.

Unfortunately, if an attacker can modify the binary file of the compiler, this is insufficient. An attacker who can control the compiler binary (directly or indirectly) can render source code evaluations worthless, because the compiler can re-insert malicious code into anything it compiles—including itself.

Karger and Schell provided the first public description of the problem. 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 first described how to modify the Unix C compiler to inject a Trojan horse (which implements a trap door), in this case to modify the operating system login routine to give him root access at all times. He then showed how to modify and recompile the compiler itself with an additional Trojan horse devised to detect itself. Once this is done, the attacks can be removed from the source code so that no source code—even of the compiler—will reveal the existence of the Trojan horse, yet the attacks would 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 implemented this attack and successfully subverted another Bell Labs group's login program as a demonstration; the subverted compiler was never released outside Bell Labs. [Thornburg2000]

For source code security evaluations to be strongly credible, there needs to be a way to justify that the source code being examined accurately represents the files actually being executed—yet this attack subverts that very claim. Internet Security System's David Maynor argues that the risk of these kinds of attacks is increasing [Maynor2004] [Maynor2005]; Karger and Schell noted this was still a problem in 2000 [Karger2000], and some technologists doubt that 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 presents a technique I call "diverse double-compiling." There appears to be no published paper discussing this technique in detail, justifying its effectiveness, or discussing the ramifications of this technique. In addition, I have not found strong evidence that this technique has actually been tried. This paper resolves these problems, including a demonstration of the technique and some lessons learned from that demonstration.

This paper begins with a background, followed by an analysis of the threat being countered and the diverse double-compiling process itself (including a more detailed description of it, its assumptions, and a justification of why it works). The next section discusses how diversity can be used to increase trust in a second compiler. This is followed by a discussion on overcoming practical challenges. The paper then presents the results of an actual example of diverse-double compiling, and closes with ramifications.

II. Background

This section describes some alternative approaches to countering "trusting trust" that are ineffective, identifies the publicly-posted information about this technique, and identifies other related work.

[1]Manuscript received May 31, 2005. Author is with (location, contact info). This work was supported by X under Grant Y, note OO, dedicated Z.

The approval of this article is pending. The article cannot be published until approved. Marking 91C57e708f3584ba6f77aceaf168db0d0b286a1c.

A.Inadequate solutions

Some alternative solutions have significant weaknesses:

1. Compiler binary files 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 difficult, compiler changes make keeping such tools up-to-date difficult, and the tool's complexity would similar to a compiler's.
3. A second compiler could compile the source code, and then the binaries could be compared automatically to argue semantic equivalence. There is some work in determining the semantic equivalence of two different binaries [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 evil. 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.

B.Diverse double-compiling

There is a little-known yet potentially practical countermeasure to this attack, which works by employing compiler diversity. In short, recompile the purported compiler source code twice: once with a second "trusted" compiler, and again using the result of the first compilation; then see if the result matches the original bit-for-bit. This countermeasure uses a second compiler to act as a check on the first. Thompson's attack implicity assumes that there is only one compiler available; by adding a second compiler, the implied assumption is broken. The trusted compiler and its environment may be initially malicious or defective, as long that does not impact their results identically, and they may be very slow. Section IV of this paper describes the technique, its assumptions, and justifies its effectiveness in more detail. I have not found any name given to this technique, so I have named it "diverse double-compiling."

There have been a few public postings about the topic in various informal locations (e.g., mailing lists). The earliest appears to be an informal posting by Henry Spencer in 1998 [Spencer1998]: "Copmile [sic] the compiler using both itself and a different compiler, yielding two binaries... Now use both those binaries to compile the compiler source again, giving two outputs. Since the the binaries are the same machinery, the outputs should be identical." Since this time, this basic idea has been posted in several places, all with very short descriptions [Mohring2004] [Libra2004] [Buck2004].

However, I have not found any peer-reviewed papers that describe or examine this technique in detail, nor I have I found public evidence that it's actually 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]

C.Related work

Lee's "approach #2" describes most of the basic process of diverse double-compiling, but he never states that if carefully controlled, the results should be bit-for-bit identical; Lee implies otherwise. [Lee2000] This is a critical component of diverse double-compiling; determining if two different binary files are "functionally equivalent" is very hard, but it is very easy to determine if two files are identical. 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]

Magdsick discusses using different versions of a compiler and different compiler platforms (CPU and operating system) to check binaries, 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 doesn't 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 binaries themselves in some environments would be pristine [Mohring2004]. He makes no notice that *all* gcc implementations might be evil, 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 binaries for "exact equivalence" unresolved.

Some effort has been made to develop proofs of correctness for compilers [Stringer-Calvert1998]; this is quite difficult even for simple languages.

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 attack 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 binaries by "randomized" transformations affecting compilation, loading, and/or execution. Their goal was to automatically change the binary (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.

Cowan et al. categorizes "post hoc" techniques – that is, adaptations to software after its implementation to improve its security. This paper categorizes techniques based on what is adapted (the interface or the implementation) and on how it is adapted (either it is restricted or it is obscured). The paper does not specifically address the problem of malicious (evil) compilers [Cowan2000].

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 [Kim1993] 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.

III. Analysis of Threat

Thompson describes how to perform the 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.

We'll begin by defining the threat. The threat considered in this paper is that an attacker may have modified binaries (which computers run, but humans don't normally view) so that the compilation process inserts different code than would be expected from examining the source code, sufficient so that recompilation of the compiler will cause the re-insertion of the malicious code. As a result, humans can examine the original source code without finding the attack, and they can recompile the compiler without removing the attack. For our purposes we'll call a subverted compiler an *evil compiler*.

Next, we'll examine 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).

A. 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, the large benefits may outweigh its costs to some attackers.

The potential benefits are immense to a malicious attacker. An successful attacker can completely control all systems that are compiled by that binary and that binary'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 victims who have highly valuable resources (sufficient to check source code) may still be 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 probably acquire limitless funds (by manipulating the entire financial system), acquire or change extremely sensitive information, or disable 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 binaries. 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 binary, 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 binaries 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. An individual entrusted with compiler development might even succumb to the temptation if they believed they could not be caught, and the legion of 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 beyond the ability of even many military organizations, and is far beyond the defensive abilities of the 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 who perform source code analysis (e.g., those who have especially high-value assets), so it is worth defending against.

B. Triggers, payloads, and non-discovery

This 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 binary has been tampered with in this way; the lack of transparency in binary 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 may reveal the existence of the attack).

This paper will refer to this attacker dilemma—that triggers may trigger when the attacker does not wish them to (risking a revelation of the attack), fail to trigger when the attacker would wish them to, or that the payload may fail to work as intended—as *fragility*. Fragility is unfortunately less helpful to the defender than it might first appear. An attacker can counter fragility by simply incorporating a number of different triggers and payloads. Even if a change causes one trigger to fail, another trigger may still fire. By using multiple 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, clock requests, 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 an evil compiler. Even if a compiler misbehaves from malfunctioning malware, the results will often simply appear to be a mysterious compiler defect; programmers may choose to "code around" the problem, in which case the real attack may be 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, evil compilers make it easier for an attacker to re-enter a previously penetrated development environment to refresh a binary 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 binary itself can influence binary 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 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.

IV. Analysis of Diverse Double-Compiling

This section describes diverse double-compiling in more detail, and presents an argument for why it counters the threat.

A. Description

Figure 1 illustrates the process of diverse double-compiling along with a self-generation check. This figure shows binary file(s) for an untrusted compiler A, binary file(s) for a trusted compiler T, and source code S that is purported to be the source code of compiler A. The shaded boxes show a compilation step; in this notation, a compilation takes source code (input from the left) and other data (input from the right) to produce a binary (output going down). If the compiler was generated from previous step, that is shown as a directed line going into the top of the box. File comparisons are shown as labeled dashed lines.

Before performing diverse double-compiling, we first do a regeneration check by taking the source code S and compiling it with compiler A, producing binary file A(S) (source S compiled by A). We then check to make sure that A(S) is the same as A. If A(S) is the same as A, then the compiler can regenerate (reproduce) itself. This does not prove the absence of malice, however.

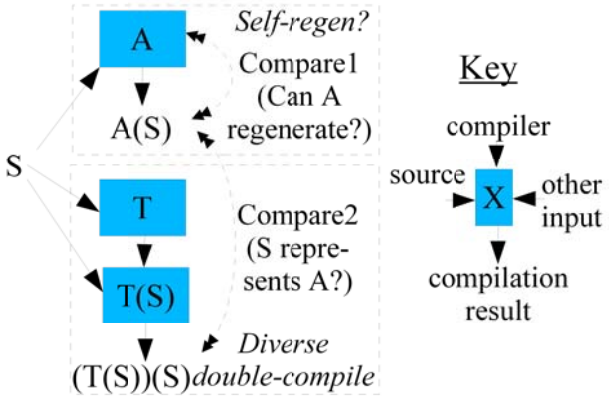

Fig. 1. Diverse Double-Compiling with Self-regeneration Check

We then perform the heart of diverse double-compiling, starting using compiler T to compile S to produce T(S) (source S compiled by T). We then use T(S) to compile S itself, producing the binary (T(S))(S). The final result is then compared (on a trusted environment) to the original A and A (S); if T(S))(S) and A (and T(S)) are identical, then we can say that S accurately reflects A (we'll see why this is so in the next section). These two compilation steps will be called stage 1 and stage 2, and are the origin of its name: we compile twice, the first time using a different (diverse) compiler. All three compilations (self-regeneration check, stage 1, and stage 2) could be performed on the same or on different environments.

B. Justification and Assumptions

To justify this, we must first state some assumptions:

1. *We must have a trusted compilation process T, comparitor, and environment(s) used in diverse double-compiling*. T, Compare2, and the environment(s) used during diverse double-compiling (that run T, S_T, and compare2, and acquire A and S), must not have triggers and payloads that affect those actions. Justifying this assumption is discussed in section V. They may have triggers and payloads, but they don't matter if they don't affect the result, and defects are likely to be detected.
2. *T must have the same semantics for the same constructs as A does, for the set of constructs in source code S.* Obviously, a Java™ compiler cannot be directly used as T if S is written in the C language! But if S uses any nonstandard language extensions, or depends on a construct not defined by a language specification, then T must implement them in the way expected by S. If a different environment is used, additional challenges may arise (e.g., byte ordering problems) unless S was designed to be portable. A defect in T can also cause a problems,

though defects will be detected by the process unless they don't affect S *or* A has exactly the same defects. Only the semantics need to be identical; T may be very slow, run on a different processor or virtual machine, and produce code for a different processor or virtual machine.

3. *The information (such as option flags) that affects the output of compilation is identical when generating A(S) and (T(S))(S).* Any input such as command line parameters (including option flags that change the results), important environment variables, libraries used as data, and so on that affect the outcome must be controlled. If timestamps are embedded in the output, ideally the timestamps should be controlled so that they will be identical in both processes; alternatives are discussed later.
4. *The compiler defined by S should be determininistic, and not use or write undefined values. G*iven the same source code and other inputs, it should produce exactly the same outputs. If the compilation is non-deterministic, it *could* be handled by running the process multiple times, but in practice it is easier to control enough inputs to make the compiler deterministic. Non-determinism hides other problems, in any case, and makes finding flaws much more difficult; uncontrolled non-determinism in a compiler should be treated as a defect. Although undefined values may be deterministic in a particular environment, if the environment changes the undefined values may too. It may be possible to work around this by carefully setting undefined values to a defined value, but it's better to fix the compiler to not do this in the first place.

The "self-regeneration" step acts as an experimental control. Until we can reproduce the binary of the compiler A using itself, we cannot hope to reproduce it with a more complicated process. This step is likely, for example, to detect non-determinism in A. If A doesn't generate itself, then we must determine how to repeatably generate A to check that process instead.

We can now make the following assertions, if the preceding assumptions are true:

1. *T(S) will be functionally the same as A, if S actually represents A.* Stage 1 of diverse double-compiling simply compiles S using T to produce program T(S). T(S) will normally have a different binary representation than A, since it was compiled using the different compiler T. Indeed, T may generate code for a completely different processor. But if source S truly represents the source of compiler A, and the other assumptions are true, then S_T will be functionally the same as A. E.g., if S is an x86 compiler, A an x86 executable, and T generates 68K code, then T(A) would run on a 68K—but since S is for an x86 compiler, T(A) would *generate* x86 code.
2. *Even if A is evil, it cannot affect the result (T(S))(S)* . During diverse double-compiling, program A (which is potentially evil) is never used at all. Instead, during diverse double-compiling we only use a trusted compilation process T and code generated by such a process T(S) in environments which we trust do not trigger on compilation of S. Thus, even if A is evil, it cannot affect the outcome.
3. *(T(S))(S) will be identical to A(S) and A iff S accurately represents A.* Since T(S) is supposed to be functionally the same as A, we can execute T(S) to compile the original source code S, producing yet another new binary (T(S))(S). But since this new binary was compiled with a program that's supposed to be functionally identical to A, and all other compilation inputs that affected compilation results were kept the same, then its output should be the same as A... and since the input is S, the output should be the same as A and A(S). In the assertion 1 example, T(A) will generate x86 code the same way A is supposed to, so if it is given S, it should produce A. If (T(S))(S) is different than A, then at least one assumption listed above is false or A has been changed in a way not visible in S (e.g., by having malicious content).

A few notes may clarify this process:

1. This process only shows that the source and binary files correspond, i.e., that there is "nothing hidden." The source code may be full of Trojan horses and errors, in which case the binary file will be full of them too. However, if the source and binary correspond, the source code can then be analyzed in the usual ways to find such problems.
2. This process only shows that a *particular* binary and source code match. There may be other binaries that contain Trojan horse(s) not represented by the source, but they will thus be different in some way.

For this technique to work, we must select a T and environment (such as the kernel, libraries being used, and so on) to do the diverse double-compiling that we can justifiably trust will not contain any triggers and payloads against S. A primary way to justify that trust is by increasing diversity, the topic of the next section.

V. Methods to increase diversity

Diverse double-compilation requires a trusted compiler T and trusted environment(s) to run diverse double-compiling, where there is a high degree of confidence that any triggers against S that may be in A will not also be present. Trust can be gained in a variety of ways, including a formal proof of compiler T's implementation and of the environments used in diverse double-compiling (making sure that the proof corresponds to what actually runs).

A simpler method to gain a great amount of trust is through diversity, and there are many ways we can gain diversity. These include diversity in compiler implementation, in time, in environment, and through mutation. The first three have been mentioned in public postings, but not examined in detail; I know of no discussion of the last point in relationship to diverse double-compiling.

A. Diversity in compiler implementation

Ideally, compiler T's binary should be for a completely different implementation. Compiler T's binary *could* include triggers and payloads for other compilers (such as compiler A), but this is less likely, since an attacker would then have to subvert the development process of multiple compiler binaries to do so.

Ideally, compiler T has never have been compiled by any version of compiler A, even in T's initial bootstrap. This is because compiler A could insert into the binary code some routines to check for any processing of compiler A (itself), so that it can later "re-infect" itself. This kind of attack is difficult to do, however, especially since bootstrapping is

usually done very early in a compiler's development and an attacker may not even be aware of the compiler T's development at that time. 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 level (e.g., to do performance analysis), so an attacker risks discovery if they subvert I/O routines.

B.Diversity in time

If compiler T and the diverse double-compiling environment were developed long before the compiler A, and they do not share a common implementation heritage, it is improbable that compiler T or its environment would include relevant triggers for a not-yet-implemented compiler (Magdsick makes a similar point [Magdsick2003]). It is possible that an attacker could arrange to include triggers in compiler A's source code once compiler A is developed, but this is extremely difficult to do, and is even more difficult to maintain over time as compilers change.

Using a newer compiler binary to check an older compiler gains less confidence; it is easier for a recently-released compiler binary to include triggers and payloads for many older compilers, including completely different compilers. Still, this requires the subversion of multiple different compilers' binaries, so even this case can increase confidence.

Diversity achieved via earlier development can only provide significant confidence if it can be clearly verified that compiler T and/or the diverse double-compiling environments are truly the ones that existed at the earlier time. In particular, old versions should not be simply acquired over the Internet without independent verification, because a resourceful attacker could tamper with those copies. Instead, protected copies of the original media should be preferred to reduce the risk of tampering. Other copies can be used 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 binary version of compiler A can be used as compiler T, if there is reason to believe that the old version is not evil or that any malicious Trojan horse in the old version of A won't be triggered by S. 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 T were truly a separate implementation.

C.Diversity in environment

Compiler T and/or S_T could generate code for a different environment or run on a different environment. The term "environment" here means the entire infrastructure supporting the compiler including the CPU architecture, operating system, supporting libraries, and so on. It should not be running any other processes (which might try to use kernel vulnerabilities to detect a compilation and subvert it). Using a completely different environment counters Trojan horses whose triggers and payloads are actually in the binaries 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 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!).

D.Mutations to create diversity

Another way to add diversity would be to use semantic-preserving mutations of compiler A's source code as the input to the first stage of diverse double-compiling; doing so would change the input as seen by T.

Semantic-preserving mutations change the source code without changing its semantics. This could include actions such as renaming items (such as variables, functions, and/or filenames), reordering statements where the order is irrelevant, regrouping statements, intentionally performing unnecessary operations that will not produce an output, changing to different algorithms that produce sufficiently similar results, and changing compiler opcode values for internal data structures. Even trivial changes, such as changing whitespace, increases diversity (these trivial changes can still be enough to counter triggers if those triggers were narrowly defined). Forrest discusses several methods for introducing diversity [Forrest1997].

By inserting such mutations, it is less likely for triggers in compiler T designed to attack compiler A will activate, and for payloads in compiler T (if triggered) to be effective. These mutations could be implemented by automated tools, or even manually. Note that if performed automatically, this implies that there is some trust given to the mutator. However, if the mutator has an unintentional defect, the result will be simply that a difference will be identified; tracking backwards to explain the difference will identify the defect insertion point, so defects in the mutator are not as serious.

VI.Practical challenges

There are many practical challenges to implementing this technique, but they can generally be overcome:

1. *Inexact comparisons may be needed*. The comparisons (Compare1 and 2) need not require an identical result as long as it can be shown that the differences that 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. An alternative that can work in some cases is to run additional self-generation stages until a stable result occurs. Another approach (used in VII) 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.
2. *Uncontrolled nondeterminism or using uninitialized data may cause different answers.* 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 to never use uninitialized data. Differences that do not affect the outcome are fine, e.g., heap memory allocations during

compilation often allocate different memory addresses, but this is only a problem if the compiler output changes depending on those addresses' specific values.

3. *It may be difficult to compile S using existing trusted compilers*. Thankfully there are many possible solutions if S cannot be compiled by a given trusted compiler. An existing trusted compiler could be modified (e.g., to add extensions) so it can compile S. Another alternative is to create a trusted preprocess step that is applied to S, possibly done by hand; as a result T would be defined as being the preprocess step plus the trusted compiler. Trusted compiler T could be created by using an existing trusted compiler (but one that can't compile S directly) to compile another existing trusted compiler that *can* compile S, i.e., the first trusted compiler is used to bootstrap another compiler. It's possible to write a new trusted compiler from scratch; since performance is irrelevant and it only need to be able to compile one program, this may not be difficult. An old version of A could be used as T, but that is far less diverse so the results are far less convincing. This also raises the spectre of "pop-up" attacks, described next.
4. *Later versions may need rechecking to prevent "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"), with the idea that defenders may not examine the source code of those particular versions in detail. Imagine that T is used to determine that an old version of compiler A (call it A-1) corresponds to its source S-1. Now imagine that an attacker cannot modify binaries directly (e.g., because they are regenerated by a suspicious user), 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-2 (which is used to generate A-2), and then remove that malevolent code from S-3. If A-2 is used to generate A-3, then A-3 may be evil, even though examining S-3 will not reveal an attack. 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. It is safer to re-run diverse double-compiling on every release (if that's impractical, at least do it periodically to reduce the window of attack).
5. *Compilers may have multiple components*. It may be necessary to break S into subcomponents and handle them separately, or in a certain order, to address dependencies.
6. *The environment of 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 diverse double-compiling technique can be used to cover these cases as well. Simply redefine A as the set of all untrusted components; this could even be the set of all software that runs on that machine (including all software run at boot time). This means that the source code for all this untrusted software is S. Consider obtaining A and S from some read-only medium (e.g., CD-ROM); do *not* trust A to produce itself (e.g., by copying files using A)! Then using a different (trusted) environment, rebuild A using S; in the limit this would regenerate all of the operating system (including boot software), application programs, and so on. If the trusted environment can regenerate the original A, then we've shown that the entire set of components included in A are represented by the entire set of source code in S. If A might have code that shrouds S, use a trusted system to view/print S when examining S.
7. *A resourceful attacker might attack the system performing diverse double-compiling (e.g., over a network) to subvert its results.* Diverse double-compiling should be done on isolated system(s). Ideally, the systems used to implement diverse double-compiling should be rebuilt from trustworthy media, not connected to external networks at all, and not run any programs other than those necessary to run the test.
8. *Few will want to do diverse double-compiling themselves.* This technique might be diffiicult to do the first time for some compilers, and in any case there is no need for *everyone* to perform this check. An organization trusted by many others (such as government agencies or trusted organizations sponsored by them) could perform these techniques on a variety of compilers/environments, as they are released, and report the cryptographic hash values of the binaries and their corresponding source code. The source code would not need to be released to the world, so this technique could be applied to proprietary software. This would allow others to quickly check if the binaries they received were, in fact, what their software developers intended to send. If someone didn't trust those organizations, they could ask for another organization they did trust to do this (including themselves, if they can get the source code). Organizations that do checks like this have elsewhere been termed "trusted build agents." [Mohring2004]

VII. Demonstration Using Tcc

Diverse double-compiling has never (to my knowledge) been publicly demonstrated, so I performed a demonstration of the technique. A demonstration requires a compiler whose source code was publicly available. For an initial test case I also wanted a compiler that was relatively small and self-contained, ran quickly (so that test runs would be rapid), had an open source software license (so changes could be publicly redistributed) [Wheeler2005] and could be easily compiled by another compiler. I also wanted a fairly defect-free compiler, since defects would interfere with these tests. For this purpose, I chose the Tiny C Compiler, abbreviated as TinyCC or tcc.

Tcc was developed by Fabrice Bellard and is available from its website at http://fabrice.bellard.free.fr/tcc/ (or via 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 very 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. 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 diverse double-compiling process, problems with casting 8-bit values and long double constants, and final results.

A.Test Configuration

I first needed to establish a test configuration. I ran all my tests 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 my purposes, and I used tcc as my simulated "potentially hostile" compiler.

I used tcc versions 0.9.20, 0.9.21, and 0.9.22 to simulate a traditional chain of recompilations; their gzip compressed tar files have the following SHA-1 values:

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

I then simulated a typical sequence of recompilations where a compiler would be updated, compiled, be used to compile itself, and then have only its binaries released.

As is usual, any such sequence must start with some sort of bootstrap of the compiler. I used gcc to bootstrap tcc-0.9.20, and had a minor challenge: gcc 3.4.3 wouldn't 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 won't accept it. Since this is only used by the intial 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 binary file which we have good reasons to believe does not have any hidden code in the binary, which can be used as a test case. Now imagine an end-user with only this binary 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 binary will be different, but this hypothetical end-user has no "pristine" file to compare against). Will diverse-double compiling correctly produce the same result?

B.Diverse double-compiling tcc

Real compilers are often divided into multiple pieces. Tcc as used here has two parts: the main compiler (binary file tcc) and the compiler run-time library (binary file libtcc1.a; tcc sometimes copies portions 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 binary file tcc is generated from the source file tcc.c and several other files; this set of source files is notated here as "tcc+.c." Note that the tcc package includes a file called tcclib which is *not* the same as libtcc1.

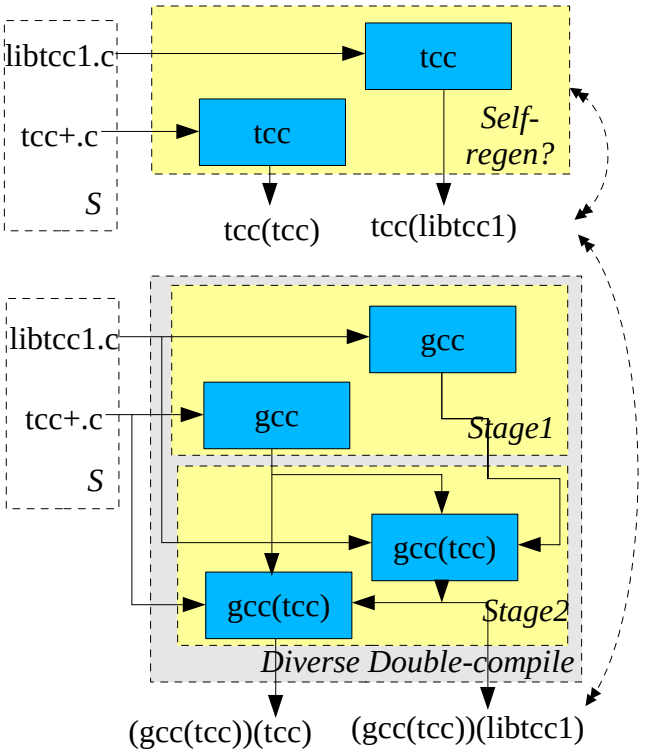

Fig. 2. Diverse Double-Compiling with Self-regeneration Check using tcc

Figure 2 shows the process used to perform diverse double-compiling with tcc. First, a self-regeneration test was performed to make sure we could regenerate tcc and libtcc1 (this was successful). Then diverse double-compiling was performed. Notice that stages one and two, which are notionally one compilation each, are actually two compilations each when applied to tcc because we must handle two components in each stage (in particular, we need to make sure the run-time is available before running a program that requires it, and that the run-time was compiled in a trustworthy way).

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 diverse double-compiling 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 I tracked this down to two problems: casting 8-bit values and representing floating point 0.0. Each of these issues is discussed next, followed by the results after resolving them.

C.Tcc defect: Casting 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. Here is a brief description of why this construct is used, why it's 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). Tcc tries to use the shorter form where possible by using statements like this (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. Tcc 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 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 I only discovered this after laboriously tracking down the problem. According to [Kernighan1998] section A6.2, converting to a smaller signed type is implementation-defined, but conversion of *that* 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):

```
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 diverse double-compiling 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.

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

D.Long double constant problem

Another problem resulted from how tcc outputs long double constants. 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. 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 10-byte long double value. However, tcc outputs long doubles as 12 bytes instead of just 10. 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.

My solution was to replace the value "0.0" with the expression `(f1-f1)`, since f1 is a long double variable known to have a finite value there (e.g., it's not a NaN). This is semantically the same and eliminated the problem.

E.Final results with tcc demonstration

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

```
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 binary 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 binaries for tcc-0.9.22 on an x86 in my environment when tcc is self-compiled a sufficient number of times to become "stable":

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

VIII.Ramifications

This paper has summarized and demonstrated how to detect Thompson's "Trusting Trust" attack, using diverse double-compiling. This 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). Unintentional defects in either compiler are detected by the process as well. The process can be easily expanded to cover all of the software running on a system above its microcode (including the operating system kernel, bootstrap software, libraries, and so on) as long as its source code is available.

Passing this test is not a mathematical proof, but more like a legal one: the test can be made as strong as you wish, by decreasing the likelihood (e.g., through diversity) that trusted compiler T and the diverse double-compiling environments also have the malicious code. A defender can easily make it extremely unlikely that an attacker could subvert this technique.

Note that this technique only shows is that the source code corresponds with a given compiler's binary, i.e., that nothing is hidden. The binary may have errors or malevolent code; this technique simply shows that these errors and malevolent code can be found by examining the source code. Passing this test *does* make source code analysis more meaningful.

As with any approach, this technique has limitations. The source code for the compiler being tested 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 (the compiler and/or the environment under test). Since the technique requires two compilers to agree on semantics, this is easier to do for popular languages where there is a public language specification and where no patents inhibit the creation of a second implementation. The technique is far simpler if the compiler being tested was designed to be portable and avoids using nonstandard extensions. It is more difficult to apply to microcode, and it does not address malicious hardware (though it can be applied to hardware specification data).

This 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 a compiler and its entire environment, and for it to be sufficiently portable so it 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, specified in public standards and implemented by multiple vendors, to implement compilers and critical infrastructure. Organizations (such as governments) could establish groups to do this testing and report the cryptographic hashes of corresponding binaries and source.

Future potential work includes examining a larger and more popular compiler (such as gcc), including an entire operating system as the "compiler A" under test, relaxing the requirement for exact equivalence, and increasing the diversity of the diverse double-compiling environment (e.g., by using a much older operating system and different CPU architecture).

References

[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

[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

[Cowan2000] Crispin Cowan, Heather Hinton, Calton Pu, and Jonathan Walpole, "The Cracker Patch Choice: An Analysis of Post Hoc Security Techniques" *Proc. of the 23rd National Information Systems Security Conference*, Baltimore, MD. June 27, 2000, http://www.scs.carleton.ca/~soma/biosec/readings/cowan-post-hoc.pdf

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

[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.

[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

[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. "Thirty Years Later: Lessons from the Multics Security Evaluation," ACSAC, September 18, 2002, http://www.acsac.org/2002/papers/classic-multics.pdf

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

[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

[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/

[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

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

[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

[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

[Spencer1998] Henry Spencer. "Re: LWN - The Trojan Horse (Bruce Perens)", Nov 23, 1998, Semi-private response (no longer on Internet).

[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.

[Wheeler 2005] David A. Wheeler, "Why OSS/FS? Look at the Numbers!, May 9, 2005, http://www.dwheeler.com/oss_fs_why.html