Challenges and Implications of Verifiable Builds for Security-Critical Open-Source Software

Xavier de Carné de Carnavalet and Mohammad Mannan
Concordia Institute for Information Systems Engineering
Concordia University, Montreal, Canada
{x_decarn, mmannan}@ciise.concordia.ca

Abstract

The majority of computer users download compiled software and run it directly on their machine. Apparently, this is also true for open-sourced software – most users would not compile the available source, and implicitly trust that the available binaries have been compiled from the published source code (i.e., no backdoor has been inserted in the binary). To verify that the official binaries indeed correspond to the released source, one can compile the source of a given application, and then compare the locally generated binaries with the developer-provided official ones. However, such simple verification is non-trivial to achieve in practice, as modern compilers, and more generally, toolchains used in software packaging, have not been designed with verifiability in mind. Rather, the output of compilers is often dependent on parameters that can be strongly tied to the building environment.

In this paper, we analyze a widely-used encryption tool, TrueCrypt, to verify its official binary with the corresponding source. We first manually replicate a close match to the official binaries of sixteen most recent versions of TrueCrypt for Windows up to v7.1a, and then explain the remaining differences that can solely be attributed to non-determinism in the build process. Our analysis provides the missing guarantee on official binaries that they are indeed backdoor-free, and makes audits on TrueCrypt's source code more meaningful. Also, we uncover several sources of non-determinism in TrueCrypt's compilation process; these findings may help create future verifiable build processes.

This article is an extended version of a paper to appear in ACSAC2014 [9].

1 Introduction

When building a software package, the compilation process generally results in different binary outputs from one compilation instance to another, especially in different environments. This leads to various problems from deduplication to trust issues. In this paper, we are mostly concerned about the security implications of such a build process, more specifically for security-critical software systems, e.g., TrueCrypt, Tor, Bitcoin Core [3], and Debian. The idea of reproducing strictly identical builds, i.e., being able to independently reconstruct the same binaries as advertised by the developers using the same source files, has been suggested only a few years ago; the oldest reference we can find is a discussion from 2007 on the Debian mailing list [47]. Several blog articles and tickets in bug tracking systems refer to this idea as machine-independent builds [7], deterministic builds [6, 36], bit-by-bit identical builds [29], idempotent builds [27], verifiable builds [44, 11], bit-perfect [12], or byte-by-byte reproducible builds [4, 5]. Some ad-hoc solutions have been discussed in these references. However, we could not locate any reference on the verifiability of open-sourced application binaries in academic literature, which we believe is a critical, but mostly overlooked aspect of current open-sourced software distribution infrastructure.

Making a build process deterministic is evidently non-trivial, as compilers and other tools in the toolchain have not been designed with this goal in mind. One of the most common sources of difference comes from embedded timestamps that reflect the time of compilation. Two compilation instances run at one second apart on the same environment may generate slightly different outputs. More difficult examples that we have found include variations linked to the type of the processor used for compilation. File checksums and cryptographic hashes differ as a result of such changes. As such hashes are used when comparing files or verifying their integrity, it then becomes irrelevant to compare hashes of different builds of the same source.

On the other hand, several existing security mechanisms rely on randomized build processes to achieve automated software diversity to restrict large-scale exploitations of software vulnerabilities. Possibly the best example of security through diversity is address space layout randomization (e.g., PaX ASLR [42]), which has been implemented in most widely-used operating systems. Software diversity has a rich history; see e.g., Forrest et al. [15], and Franz [16] (also the recent survey on diversity [28]). By definition, software diversity mechanisms are mostly adverse towards deterministic builds. Other sources of diversity in binary code may include compiler optimization techniques, especially the ones independent of source code, e.g., profile-guided optimization (PGO [32], recently adopted by Mozilla Firefox [34]).

The importance of deterministic builds has gained general awareness after classified documents from the U.S. National Security Agency (NSA) have been leaked by Edward Snowden starting in June 2013. The magnitude and depth of the intelligence programs explained in these documents suggest, among other things, the ability of the NSA to perform surreptitious man-in-the-middle attacks to infect targeted systems, or more generally make those systems download malicious software (cf. NSA TURBINE [17] and the Dual_EC_DRBG case [30]). It then becomes easy to insert malicious behavior such as backdoors in binary files that are downloaded by a majority of users, particularly on the Windows platform. Man-in-the-middle attacks can be prevented by the use of TLS-encrypted communications. However, such protection does not prevent a malicious or coerced software author from distributing malicious binaries that do not match the available source code. Moreover, an author can digitally sign her software package, e.g., via an embedded X.509 certificate on Windows executables; however, this may constitute a single point of failure (cf. Flame malware [21]). The ability to verify both the source code and the corresponding executable against a consensus could prevent such attacks on downloaded software.

Security-sensitive applications such as Bitcoin Core and Tor have started using a deterministic build approach, to enable decentralized trust through multiple independent signatures from users who compiled the application and submitted signed hashes of the output. Both Bitcoin Core and Tor are based on Gitian [18], a Ruby and bash wrapper for Ubuntu's python-vm-builder that was created in 2009 to ensure the build security and integrity of Bitcoin. In 2013, the Tor Browser Bundle [36] adapted Gitian for its purpose, which involves the cross-compilation of the bundle for Mac OS X and Windows from a Linux environment. However, several aspects for a deterministic build remain unaddressed.

As a case study, we focus on TrueCrypt [45], an open-source tool capable of on-the-fly encryption.[1] It creates encrypted containers at different levels (file, partition and entire disk), provides a full disk encryption (FDE) feature under Windows with pre-boot authentication, and even offers plausible deniability of hidden partitions. TrueCrypt is available as compiled binaries for Windows, Linux and Mac OS X, along with its source code. However, as we mentioned, simply compiling the source code is not enough to replicate a match with the given binaries. Hence, reviews and audits of its source code usually discard this difficult verification step, and may qualify TrueCrypt as secure, regardless of whether the audited source corresponds to the targeted compiled application that is distributed and used by the majority of users. Our analysis brings this missing connection and makes the conclusions of those audits more meaningful. Similar to other security software, such as Tor, TrueCrypt also appears to be a perfect target for government surveillance (see [17]). A legitimate question then arises: Are the binaries provided on the website different than the available source code and do they include hidden features such as a backdoor? In this paper, we provide the answer to this question and make the following contributions:

1. We recompiled sixteen versions of TrueCrypt from v5.0 to v7.1a and analyzed the differences between our builds and the official ones. These versions date back from February 2008 up to the latest fully-functional version released in February 2012.

2. We detail the major challenges we faced to replicate a close match, and successfully explain the remaining differences, if any. We then conclude that all of TrueCrypt's signed binaries directly come from their respective sources and no backdoor has been inserted in the binaries.

3. We identify several key sources of non-determinism to be taken into account to realize deterministic builds. Our methodology can also help verify other applications that do not provide a deterministic build.

4. Finally, we summarize the lessons learned from other on-going projects that aim at achieving deterministic or reproducible builds, and their limitations.

[1]Even though recently the TrueCrypt developers announced the discontinuation of their work, other groups are joining effort to keep such an essential tool alive; see, e.g., `http://truecrypt.ch`. The TrueCrypt audit project (`http://istruecryptauditedyet.com`) also remains active.

2 Definition, threats and challenges

2.1 Definition

The general idea behind a deterministic build is to record the environment when building the official release of a project, then replay the behavior of this environment in later builds to achieve the same results. This process removes sources of non-determinism that are out of control in a regular building process. A broader term has been coined by Debian as reproducible build [10], which emphasizes more on replicating the official build, regardless of the process involved. To reconcile existing terminologies, we suggest the following definition:

A build is verifiable if any two instances of the build process produce identical results. This can be achieved through a deterministic process, in which case both builds are byte-by-byte identical (and hence the process is machine-independent); or by matching the builds at a higher semantic level (e.g., by ignoring unimportant differences).

2.2 Assumptions

We assume that the compiler is trusted; cf. Thompson [43] (see also [51] for a proposal addressing the untrusted compiler problem). We also trust the operating system (OS), as it would make no sense to trust a regular program (e.g., a compiler in this case) running on an untrusted OS. The hardware platform including the CPU of the build system is also trusted (but see e.g., [26]).

We also make the following assumption regarding the multiple independently signed hashes of recompiled binaries. If the verification process succeeds by consensus (i.e., the majority of the signed hashes match the hash computed by the user), we assume that malicious signatures cannot represent a majority in the list of signatures. This assumption does not hold in situations where a powerful attacker can compromise a significant number of signatures (i.e., beyond the majority threshold) with ones that correspond to malicious binaries. Other verification schemes could be considered (e.g., see Perspectives [50]). Finally, note that deterministic builds are only interested in matching sources with binaries; hence, we do not make any assumption regarding the trustworthiness of the source, which remains under the scope of source-code audits.

2.3 Threats considered

Our primary consideration is that users do not recompile software packages from the source. Based on it, we consider the following threats and explain how a deterministic build prevents against them.

Targeted attacks on binaries. We assume that an attacker can alter the binaries received by a targeted user (e.g., a coercive government against a Tor user). Such modifications will remain undetected if the integrity of the delivery channel or an author's signing key is compromised. This attack is applicable to TrueCrypt, as it could only be downloaded through an insecure channel without a TLS-encrypted connection. A deterministic build provides a match between the source of an application and its compiled version distributed to users. The match operates at the file level on the output side, enabling hash comparison of output files and verification of independent signatures; such verification enables decentralized trust. Thus, a user can simply compare the hashes of her copy of the application against the independently signed hashes, and identify whether she has been subject to a targeted attack.

Untrusted authors. We assume that an untrusted open-source developer (coerced or malicious) in charge of compiling the official build, provides backdoored binaries for distribution through the official channel (e.g., website or update server), but leaves the available source untouched. If targeted attacks on binaries are addressed, developers can no longer provide binaries that do not match the source, since the official signatures would differ from any independent ones generated by recompiling the application from its source. Addressing targeted attacks on binaries hence provides the side effect of protecting against untrusted authors.

Targeted attacks on the source. We assume that an attacker can alter the source obtained by a targeted user. Such type of attackers can include the developers of an application, who wish to mislead a source reviewer (as in the case of a security audit). If targeted attacks on binaries are addressed, a user wishing to recompile the application from the source benefits from an additional feature: she can also verify that the source code she obtained corresponds to the official binaries that have been independently signed. Indeed, if the hashes of the recompiled binaries do not match the ones that are independently signed, the user will detect that she has obtained a different source. The granularity of the detection of mismatching inputs may vary (i.e., a single file or a group of files), since mismatching binaries can be compiled from several input files.

Targeted attacks on both the source and binary files. In a particular case of targeted attack, we assume an attacker can alter both the source and binaries for a specific user through a man-in-the-middle attack, or through the official delivery channel. Such a user could then be tricked into thinking that she has verified the official build by recompiling the source herself and matching her build with the official downloaded binaries. Even though a TLS-encrypted channel provides better security against a man-in-the-middle attack, it does not fully prevent against this threat (e.g., if altered files are made available through the official channel for a particular user). Multiple independent signatures, enabled by deterministic builds, reduce the probability of such attacks, since the difficulty to compromise several keys would also increase.

A deterministic build thus bridges the gap between sources and binaries in both directions, i.e., it allows any user to match binaries to reference sources, and to match sources to reference binaries. Also, it does so without involving average users into any technical details of the compilation process.

2.4 Verifying non-deterministic build

Context. Supposing that all future open-source software follow a deterministic build process, one may still be concerned about past software packages that provide compiled binaries but were not compiled with verifiability in mind. In this case, the problem is to match the sources with the available binaries that are (ideally) signed by the developer, or provided through an integrity-protected channel to consider them as official. TrueCrypt falls in this category, which relates to the untrusted authors threat.

Feasibility. A naive approach to achieve the verification of past software is by manually inspecting each version of a selected application and replicating a close match, then explaining any remaining differences. A custom build process could be created for a specific software instance to make it deterministically buildable. It would probably be version-specific, as supporting several versions and handling many potential environments would require designing a universal deterministic builder. However, manual review of past software is painstaking, as illustrated by our case study in Section 3, and adapting a deterministic build that exactly matches the official binaries may be sometimes impossible.

Automated verification of past software can also be very challenging. In case the required resources (e.g., the compiler information, project configuration or building steps) are unavailable or not properly identified, one would need to leverage compiler fingerprinting techniques (cf. FLIRT [22]) to identify the optimization level and other options passed to the compiler, along with the correct dependencies. In addition, software-specific differences (e.g., TrueCrypt installer's custom checksum, see Section 3.3.3) may prevent generalization of an automated verification technique for such non-deterministic builds.

Note that verifying future deterministic builds and past software packages do not share the same requirements. For packages whose output is not deterministic, the equivalence with the official build needs to be proven for every build, while waiting for newer versions that will hopefully leverage a deterministic build process. This situation also highlights the fact that a build does not need to lead to an output that is byte-by-byte identical with the official build to be verifiable. Superficially variable areas (e.g., timestamps) can be ignored by a high-level comparison, as opposed to a simple file hash comparison. As we observed in our analysis of TrueCrypt in Section 3, the remaining differences we faced after setting up a proper environment were found to be legitimate and explainable, as opposed to malicious differences.

3 Case study: TrueCrypt

TrueCrypt does not provide any explicit way to verify its compiled binaries as made available for Windows, Linux and Mac OS X. Anyone wishing to compile the sources will get binaries different than the official ones, as identified by others in the past (see e.g., [38]). This has led to some speculations regarding the possibility of having backdoors in the official binaries that cannot be found easily as they would not be apparent from the source. This concern has also been raised in a security analysis of TrueCrypt by the Ubuntu Privacy Remix Team [46], in which the authors conclude that they cannot link the result of their code analysis to the official binaries because it would require "a very expensive reverse engineering".

In this section, we present the challenges we faced to compare all the official releases of TrueCrypt for Windows since February 5, 2008, including versions 5.0, 5.0a, 5.1, 5.1a (and its second release), 6.0, 6.0a, 6.1, 6.1a, 6.2, 6.2a, 6.3, 6.3a, 7.0, 7.0a, 7.1 and 7.1a (February 7, 2012) with the alleged corresponding sources. An analysis of version 7.1a was also documented by the first author in an online report [8];[2] we reuse parts of that report here. We present the challenges in the order as we faced, to highlight the difficulties posed

[2]The article was discussed in a Slashdot post; see: `http://it.slashdot.org/story/13/10/24/169257/how-i-compiled-truecrypt-for-windows-and-matched-the-official-binaries`.

by the sources of non-determinism, and the reasoning we followed to get down to the root cause of them.

3.1 Our test environment

To replicate a clean environment with the control over which compiler and tools are installed, we leverage the snapshot feature of VMware Workstation. It enables the creation of virtual machines snapshots that we can later fork to a different path. This feature is useful when installing multiple applications in a series, as we can take a snapshot after the installation of each of them, and later backtrack to a particular snapshot to continue installing different versions of the remaining tools in the toolchain. We configured three virtual machines: one running Windows 7 Professional 64-bit edition and the two others running Windows XP Professional 32-bit edition; in total, we created 45 snapshots, which took about 175GB of disk space. For our experiments, we mostly use a PC with an Intel Core i5-2400 processor; we also performed some tests on another machine with an AMD FX-8350 processor to confirm the origin of yet another source of non-determinism involving the CPU manufacturer.

3.2 Preparing the environment

3.2.1 Prerequisites

In the latest version of TrueCrypt's source package, the `readme` file specifies the following list of software, tools, SDK and additional files as requirements (as quoted from the file): (a) Microsoft Visual C++ 2008 SP1 (Professional Edition or compatible); (b) Microsoft Visual C++ 1.52 (available from MSDN Subscriber Downloads); (c) Microsoft Windows SDK for Windows 7 (configured for Visual C++); (d) Microsoft Windows Driver Kit 7.1.0 (build 7600.16385.1); (e) RSA Security Inc. PKCS #11 Cryptographic Token Interface (Cryptoki) 2.20 header files; (f) NASM assembler 2.08 or compatible; and (g) gzip compressor.

For the older versions of TrueCrypt we studied, older versions of similar tools are required. Going back to version 5.0, we need to gather Microsoft Windows Driver Kit (WDK) for Windows Vista (build 6000) and version 7.0.0 (build 7600.16385.0), Microsoft WDK for Windows Server 2008 SP0/SP1 (build 6001.18001/2), Microsoft Visual C++ 2005 SP1 Professional Edition, NASM assembler 0.99 and 2.06, along with Yasm assembler (no version specified).

The first problem in creating the build environment is that it requires compilers and resources that can be difficult to find or are non-free; notably, Visual C++ 1.52 that was released 20 years ago (in 1994) and is only available to MSDN subscribers (which membership costs at least US$1,200 at the time of the writing), or for Microsoft Certified Trainers (MCT). In our case, we first had to search for copies online before being provided with the original file by an anonymous contributor with an MSDN subscription, in reaction to our online report. Indeed, the academic access to MSDN (MSDNAA) of our university was not privileged enough to access such old software. For Microsoft Visual C++ 2008 (Visual Studio), one also needs to have an MSDN subscription since it is now an old version that is no longer publicly available as a trial version on Microsoft's website (although a direct link to the ISO file on Microsoft's servers can be found by crawling the web). This software was accessible via our MSDNAA access.

Second, Microsoft WDK 7.0.0 and below are no longer available at Microsoft at the time of the writing (May 2014) because of newer versions that superseded the previous ones, but was available at the time of the first analysis (October 2013). This also highlights a general problem for verifying even relatively new software, as current dependencies may become permanently unavailable at any time. Additionally, there exist three versions of the Windows 7 SDK: version 7.0 for .NET Framework 3.5 SP1, v7.0A included in Visual Studio 2010 only, and v7.1 for .NET Framework 4.0. Also, we do not know *a priori* to any test whether a different version than the one in the authors' environment can lead to changes in the compiled output. Fortunately, is it still possible to find direct links to download old SDKs at Microsoft. For old WDKs though, only MSDN subscribers can obtain legal copies, which pushed us to search for other channels due to limitations of our academic access to MSDN. We wish to thank anonymous contributors for providing us with the missing pieces that we could not find. We verified the hashes of the files we received against the official hashes published on the MSDN website.

Finally, the RSA PKCS #11 Cryptographic Token Interface (Cryptoki) 2.20 header files and NASM assembler 2.08 are freely available online. The gzip compressor is also available for Windows thanks to the GnuWin project [19]. The version number used by the developers is not mentioned. However the latest GnuWin's gzip version (1.3.12-1) dates back to 2007, so we assumed that this version or a compatible one was used by the original developers. A different compressor or version of that compressor can lead to a different compression algorithm or file format and result in a different output. The GnuWin's version of gzip (1.3.12-1) fortunately worked for our purpose. We later found in the source code that `gzip.org`'s release was suggested by the developers.

Table 1: File names and sizes from our first compilation attempt (VS2008 SP1 without updates) vs. the original ones

File name	**Sizes in bytes**	
	Our build	Official build
TrueCrypt.exe	1,507,840	1,516,496
TrueCrypt Format.exe	1,602,048	1,610,704
truecrypt.sys	224,128	231,760
truecrypt-x64.sys	223,744	231,376
TrueCrypt Setup.exe	1,056,768	N/A
TrueCrypt Setup 7.1a.exe	3,432,471	3,466,248

Although not mentioned in the TrueCrypt's project `readme` file but pointed out in [40], the *dd* tool [23] is also required during the build process. Some dd ports for Windows do not behave correctly during the compilation process (e.g., different arguments are expected and no output is generated). We first used an incompatible version (from `chrysocome.net`) that achieved incomplete builds. We later found a working version in the CoreUtils package for Windows [19].

3.2.2 Initial challenges

Once the environment is correctly installed, we can open the Visual Studio solution file *TrueCrypt.sln* and build it in "release" configuration.

Flat comparison. Our first naive attempt takes place in a virtual machine running Windows 7 Professional 64-bit edition with the prerequisites installed. Table 1 shows the sizes of our compiled binaries and the official ones from v7.1a.

File sizes do not match, leading to different hashes. If the build process was made deterministic, we would have a match at this point. First of all, one obvious reason for the file size difference is the presence of digital signatures (or lack thereof) that we explain below. However, there are also a number of other differences that we explore in this section.

The official binaries are all signed with an embedded X.509 certificate that belongs to the "TrueCrypt Foundation", which increases the file size. This is an issue for the purpose of reproducing an exact match, since it is impossible to generate a signature for the binaries on the developers' behalf. Hence, it is not possible to reproduce an exact match of any of the binaries if we are not in possession of the official build's files. Indeed, the original signatures can be extracted from the official files and reused in our build as is. These signatures would be valid if the rest of the files also matched. Also, note that the installer (TrueCrypt Setup.exe) is incomplete after compilation. We must run it with a special flag (`-p`) to package the other binaries together and output a complete installer named TrueCrypt Setup 7.1a.exe. In Table 1, we packaged our own binaries for comparison. To replicate an exact match, the binaries should be signed prior to being packaged. We therefore ignore the installer until we can reproduce the other binaries.

Visual Studio updates. The digital signature represents a small fraction of the binary files (7,631 bytes in each file in v7.1a), and is placed at the end of the files, which makes it easy to locate. However, we found that a byte-by-byte comparison of each file still produces significant changes, beyond the signature. We then considered an advanced comparison technique that attempts to identify matching blocks of bytes, even if their sizes slightly differ. Such comparison generally yields a score that represents more accurately the real content difference. With respect to the official file sizes and using such a comparison technique,[3] we find 60,049 (4.0%) of mismatching bytes in TrueCrypt.exe, 34,746 bytes (2.2%) in TrueCrypt Format.exe, 7,673 bytes (3.3%) in truecrypt.sys, and 7,878 bytes (3.4%) in truecrypt-x64.sys.

We further investigated additional patches available for Visual Studio 2008 SP1 that do not change the version number of the software but lead to major changes in the output. In other words, we should restore an environment that contains the same updates for the compiler as what the authors had installed when they compiled the official release. As there is no mention about which patches were installed in their environment, we needed to manually go by trial and error. To reduce the differences between our build and the official binaries, we figured that we needed to install all the updates that were available when a particular version of TrueCrypt was released. The core updates are KB971092, KB973675, and KB972222 for versions 6.1 to 7.0a, with the addition of KB2538241 for 7.1 and 7.1a. With these only updates installed, we minimize the differences. Figure 1 shows a visual representation of the differences between our build and the official files in the case where no updates are installed, or the proper updates are installed. This indicates that the developers had their system up-to-date, however such observation may not stand in general.

For versions 5.0 to 5.1a that were compiled with Visual Studio 2005 SP1, a single core update (KB937061) is available, which was released five months before the release of TrueCrypt v5.0. We hence installed it before carrying our experiments, unlike our first tries with more recent versions of TrueCrypt. For versions 6.0 and 6.0a, no update was available for Visual Studio 2008 at the time of their release.

Thus, for the purpose of verifying official builds, it is important to use the same exact version of the compilers and tools installed on the developers' build machine, since a slight difference can significantly change

[3]We used Beyond Compare v3 (`scootersoftware.com`).

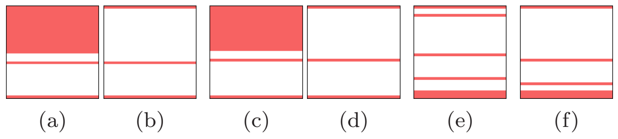

Figure 1: Visual differences on the binary files between our build and the official one for TrueCrypt 7.1a using an advanced comparison technique. (a), (c), (e) and (f) represent binaries compiled without any updates for VS2008 SP1 in our build while others are compiled with the proper ones. Files in (e) and (f) are not influenced by such updates. (a) and (b) represent TrueCrypt.exe; (c) and (d) TrueCrypt Format.exe; (e) truecrypt.sys; (f) truecrypt-x64.sys. Red areas represent approximate differences with coarse granularity (the thinnest red line represents up to 2.8% differences).

the output binaries.

3.2.3 Build path

The files truecrypt.sys and truecrypt-x64.sys are the 32-bit and 64-bit Windows drivers, respectively, which take care of all features related to the OS, such as providing virtual disks and supporting full disk or system partition encryption. The number of differences between our build and the official one remains significant. While investigating them, we found that these files contain various debug information, including the full project path. These debug parameters are references to a Program DataBase (PDB) debug info file and are inserted by the linker to match the corresponding debug file [33]. The short debug section in the binaries contains the path of the PDB debug file, located under the project directory. The project folder in our test environment was located on the user's desktop while the developers had it apparently in c:\truecrypt-7.1a. The difference in the path length leads to a shift of the remaining data in this debug section, plus an offset in some addresses that point to locations in the file. By compiling the software again after moving the project directory to the same location as in the official build, these differences were gone. Note that, in the other files, there is no debug information and hence no link to the build path. Hence, the build path can sometimes influence the output and should be taken into account while setting up the environment. Throughout the 16 versions we analyzed, the project path usually contains the version number and is located in the root of the C drive; however some project paths are found without a reference to the version number and/or are located in an E or T drive. As we found, the project paths are c:\truecrypt-7.1 for version 7.1, c:\truecrypt-7.0a for 7.0a, c:\truecrypt for 6.3a, t:\truecrypt for 6.3, c:\truecrypt-6.2a for 6.2a and similar down to 5.1a, e:\testworkspace\truecrypt\main for 5.1, 5.0a and 5.0.

3.3 Sources of non-determinism

To understand the differences between our compiled binaries and the original ones, we perform a byte-by-byte comparison. Below we analyze each file individually. We mainly focus on the latest version of TrueCrypt (v7.1a) with all addresses and offsets cited referring to this version, unless otherwise specified.

3.3.1 TrueCrypt.exe and TrueCrypt Format.exe

Visually from Figures 1 (b) and (d), there are three main regions that host differences in TrueCrypt.exe.

First region. The first region is located in the file header, and contains three differing elements. The first one is at file offset 0x000000F8 that corresponds to the time/date-stamp in the COFF/PE file header. The second is located at file offset 0x00000148 that corresponds to the checksum in the PE optional header. Finally, the third one at offset 0x00000188 that corresponds to the certificate table in the optional data directories header. According to Microsoft documentation [31], time/date-stamp corresponds to the time and date the file was created. Since we compiled TrueCrypt at a different time than the developers, this difference is legitimate. Then, the checksum corresponds to the image file checksum based on a Microsoft-specific algorithm [49]. This checksum is different because our compiled executable has slight differences that are covered by this checksum, resulting in different values, for which we verified the correctness. Hence, this difference is also legitimate. Finally, the certificate table contains a certificate data field, which is a pointer to a certificate in the file; along with field about the certificate size. This table provides information regarding the X.509 signature over the file that is included in the official binaries. Because we do not have certified binaries, the certificate table is all zeros, whereas the original file points to some certificate data at offset 0x00170600 in the file (see the third region).

Second region. The second region of differences is located at about two thirds of the file. It corresponds to a readable time and date written in English and what seems to be also a timestamp encoded as a 32-bit integer. Converting this alleged timestamp to a readable date and time, we find that it matches the written date, confirming it is simply a timestamp. Interestingly, we can deduce the time zone of the compiling machine, since the time written is interpreted with respect to the machine settings while the timestamp is a POSIX time representing the number of elapsed seconds relative to a reference UTC time. In this case, the timestamp of the official binary converts to 2012-02-07 09:08:50 UTC while the written date and time reads "Tue Feb 07 10:08:49 2012". This informs that the time zone of the machine which built the executable was set to UTC+1 (CET).

In the 11 versions from 5.0 to 6.3, more differences arise in this section. Further investigations reveal that they are located in the resource section of the executable, and correspond to an Interface Definition Language (IDL) file compiled by the Microsoft IDL compiler. The OLE-COM Object Viewer in Windows SDK can decompile this resource. By comparing our decompiled IDL resource to the decompiled official one, only the timestamp and written date remain different, meaning both resources are functionally identical. Despite our efforts to recreate many possible build environments, we were unable to reproduce an IDL resource that only includes differences in the timestamps in a way that is visible from the compiled format directly, as it is the case in later versions of TrueCrypt.

For version 6.3, using the SDK for .NET Framework 3.5 SP1 (v7.0) that was available at the time of release of this version of TrueCrypt, yields a different Rich signature (see [37]) in the DOS stub part of the PE header of TrueCrypt's executables. The difference resides in the *comp.id* row number 148 that corresponds to a subversion 30729 in our case, while the original file shows a subversion 21022. This indicates that a component in the toolchain differs. Once we switched to the latest version (v7.1), this problem was solved and only the difference in the IDL resource remained. However, this later version of the SDK was released 7 months after the release of TrueCrypt 6.3, and thus could not have been used by the developers. One possible explanation is that the authors had Visual Studio 2010 installed on their system, which comes with the intermediate version 7.0A of the SDK, and configured Visual Studio 2008 to use it. However, Visual Studio 2010 was also released 6 months after this version of TrueCrypt. The only realistic possibility is that the authors were using Visual Studio 2010 beta 1 or 2 on their machine, which was released 5 months and 2 days, respectively, before TrueCrypt 6.3. They would have had to configure Visual Studio 2008 accordingly. We were unfortunately unable to find this beta version for further tests. This problem demonstrates that the sources of non-reproducibility can be difficult to identify and thus difficult to troubleshoot.

Third region. Finally, at the end of the file, the third region of differences starts at 0x00170600 and shows us that the original file contains more information. Recall from the first region that this offset is directly pointed to by the certificate table and hence is related to the digital signature. We can safely ignore the presence of the certificate in the official binaries, since a signature and certificate are normally harmless (cf. Microsoft's documentation: "[t]hese certificates are not loaded into memory as part of the image." [31]) Thus, if this section contains malicious code, it has to be loaded by the program first, which would be seen in the source code (its review is out of our scope, though).

It is to be noted that apart from these three unimportant mismatches (timestamps, checksum, presence of certificate), the rest of the files are identical. This indicates that our TrueCrypt.exe and the official one are identical. Also, in TrueCrypt Format.exe, we find the same patterns of difference as TrueCrypt.exe. As we explained the unimportance of these differences in the case of TrueCrypt.exe, we can conclude that our TrueCrypt Format.exe and the official one are also coming from the same source.

3.3.2 Truecrypt.sys and truecrypt-x64.sys

Apart from the build path issue that we solved when creating the build environment, other differences remain.

In the 32-bit driver, a difference at file offset 0x00000270 corresponds to the time/date-stamp in the PE headers. File offsets 0x0001EA44 and 0x00034184 show the same timestamp difference. More specifically, the one at 0x0001EA44 matches the timestamp location in the debug directory structure [31], which contains the address of the debug section. In turns, this debug section is located at file offset 0x0002CBA8 in the 32-bit driver and offset 0x0002F490 in the 64-bit driver, where another difference is found (see the next paragraph). Similarly, the timestamp at offset 0x00034184 matches the location of the timestamp in the export directory table. File offset 0x000002C0 is the optional PE checksum header, which also differs for the same reason as in the previous files: the file has slight differences that lead to a different checksum. File offset 0x00000300 represents the certificate table difference, which as we explained is expected. The end of the original file contains the certificate.

In the PDB debug section, pointed by the debug directory previously mentioned, we still find one difference of 16 bytes starting at 0x0002CBAC. The format of this section is undocumented, however it is explained in [20] as containing a signature (the string "RSDS"), followed by a GUID (Globally Unique IDentifier) that is regenerated in each build and is used by the debugger to link a binary file to its PDB debug file. Our 16-byte difference matches the location of the GUID. Next to it is an "age" field followed by the path of the debug file, which led to a difference we had previously resolved by using the same project path as in the official build. As the project was compiled in release configuration (not in debug configuration), such information should not be present at all in the output files. Their presence remains unclear.

In versions 6.3 and 6.3a, the `readme` file suggests the use of NASM "version 2.06 or compatible". However, if we use version 2.06 during the build process, NASM will crash when assembling the 64-bit driver. While investigating this problem, we found a ticket on the NASM bug tracking system [41] mentioning this specific issue. The issue has been resolved in 2.08-rc1, which was released one week before TrueCrypt v6.3's realease. One may further correlate this bug ticket with the internal development of TrueCrypt.

The 64-bit version of the driver, truecrypt-x64.sys, shows the same patterns. However, in version 6.2 and 6.2a, we observe an additional difference spanning on 5 consecutive bytes at file offset 0x0001CFCB. Contrary to other differences that we identified so far, this one does not affect metadata but rather is located in the `.text` section of the binary file, which contains the logic of the driver. Our build reads `0F 1F 44 00 00` while the official build reads `66 66 90 66 90`. The disassembled binary files show a more comprehensive comparison shown below.

```
66 66 90          data32 xchg ax,ax
66 90             xchg   ax,ax
0F 1F 44 00 00    nop    DWORD PTR [rax+rax*1+0x0]
```

Functionally, both sequences are effectively realizing No Operation (NOP). These NOPs only serve alignment for the remaining code. We can find a partial explanation for this difference in Intel's documentation [24], which lists sequences of various lengths that realize a NOP (in Table 4-9 entitled "Recommended Multi-Byte Sequence of NOP Instruction"). Our NOP corresponds to a 5-byte NOP in this table. However, we cannot find the official build's NOP sequence in this table. The explanation actually lies in the type of processor used in the build environment. While Intel's documentation recommends `0F 1F 44 00 00` for a 5-byte NOP, AMD's documentation [1] recommends `66 66 90 66 90`. We can infer that the compiler determines the processor it is running on and adapts its output accordingly. This hypothesis is confirmed after we compiled this version of TrueCrypt on a machine with an AMD processor. The 5-byte NOP was present as in the official build. We can further deduce that the developers were using an AMD processor for the release of versions 6.2 and 6.2a.

3.3.3 TrueCrypt installer

Now that the remaining files have been analyzed and the differences between our build and the official one have been explained, we can package the original files with our compiled installer. After packaging the original files with our compiled installer, we obtain an installer of 3,458,614 bytes, which is close to the original installer's size (3,466,248 bytes). Again, the usual time/date-stamp, checksum and certificate table differ, and the original installer has a certificate at the end of its file. A new difference occurs at 0x0034C632 on 4 bytes that look like a checksum. By investigating TrueCrypt's source code, we can find that it is indeed a checksum. During the packaging of the files, the installer computes an integrity checksum over its complete version. At this point, the installer is not yet signed, which means it does not embed a certificate and its certificate table is all zeros. However, after the installer is signed, it itself needs to be able to recompute the checksum over its unsigned version. To achieve this, the installer computes the checksum by replacing bytes between file offsets 0x00000130 and 0x000001FF (inclusive) with zeros, so as to zero out the Certificate Table. This range however contains more information than just the certificate table, specifically half of the optional PE header and the whole data directories, including the Export, Import, Resource, Exception, Base Relation, and Import Address tables. In practice, erasing this byte range effectively deletes these fields in the header, and weakens the coverage of the integrity check. The installer also truncates the file after the magic word "TCINSCRC", which is located right before the checksum in question, effectively deleting the digital signature. The CRC32 computed over this modified file corresponds to the alleged checksum we are investigating.

Version 5.1a was released twice; the second edition was released the same day as version 6.0. This second edition repackages the original files with a new installer derived from the installer of version 6.0. The main apparent goal was to update the license shown during the installation. Comparing this new installer with the installer from v6.0 however shows significant differences. Once disassembled, many addresses differ, some part of the code is changed (mostly for alignment), and eight resources are completely different. As the exact source of this new installer was never made available, we are left with the option of reverse-engineering the installer or trying to recreate its source, based on the knowledge that it is an intermediate version between 5.1a and 6.0. After several experiments, we concluded that this installer comes from v6.0 with changes made to VERSION_STRING and VERSION_NUM constants to simulate version

5.1a. Once compiled, the installer packages the original files from v5.1a with the new License.txt file. Incidentally, when using this new installer on a system-encrypted environment, the bootloader installed by the installer is in fact the one from v6.0 renamed as v5.1a (which caused the differences in the resource section as several versions of the bootloader exist and they are gzipped in that section). This brings another source of non-determinism, which is any (allegedly unimportant) code modification that the authors made to compile a build without releasing or documenting them.

3.4 Summary

In this analysis, we showed that the compiled versions 5.0 to 7.1a of TrueCrypt are directly compiled from the available source code. From a security perspective, this shows TrueCrypt is not backdoored by the developers in a way that is not visible from the sources, i.e., the provided executables for Windows are not added with any feature not visible from the sources; addressing the untrusted authors' threat.

From a software engineering perspective, our study helps identify sources of non-determinism in a simple Visual Studio project over the years, from which many findings can be generalized. We now summarize our experience throughout the 16 versions of TrueCrypt. The most important factor is the version of the compiler and of the additional resources required to build the project. We showed that a slight variation in the version number (or minor update) can significantly change the output. Then, we revealed a particular feature in the Microsoft Visual Studio compiler that optimizes the alignment of NOPs according to the recommendations of the manufacturer of the processor on which the compiler is running, making the build processor-dependent. We showed that the project path is sometimes stored in the output and results in cascading variations due to a shifting effect. Moreover, the instructions provided by the authors of the software may not be accurate and lead to incomplete builds as we observed with a bug in the version of NASM advertised in the `readme` file. These instructions may not be precise enough to avoid ambiguities in version numbers as we experienced with Visual Studio or Windows SDK; or, they may be incomplete altogether. In our case, the dd utility was missing. Incomplete code may also be an issue, as we saw with the second version of v5.1a's installer.

Furthermore, embedded timestamps proved to be a very common source of variations, leading to checksum differences. We also showed that the timestamps and checksums may not be included only once in a pre-defined location (as in the PE headers), but also in the resource section of the binary and they may follow application-specific algorithms (as in the case of TrueCrypt's installer). Debug information embeds unique identifiers that are randomly generated during every build. Finally, as embedded digital signatures are impossible to regenerate, they should either be copied as is from the official build after investigation of their content, or reside in a separate file to enable file-based comparison. We emphasize on the fact that recompiling a project twice on the same environment does not necessarily exhibit all sources of non-determinism. One needs to try on different environments with various configurations (e.g., heterogeneous CPUs and other hardware) to see all the differences, if any. Surprisingly, in our analysis, we did not encounter any differences caused with a different operating system (we compiled some versions on both Windows XP and 7).

4 Towards deterministic builds

In this section, we discuss efforts from other projects involving deterministic builds, and summarize their findings and lessons learned.

4.1 Gitian for Bitcoin

The most advanced work to date on deterministic builds is probably the one initiated by Bitcoin, named Gitian [18], and later adapted by the Tor project for the Tor Browser Bundle. Gitian provides a virtual environment in which various sources of non-determinism can be fixed. It is essentially based on the python-vmbuilder package that builds a Ubuntu virtual machine, and wraps it with several scripts. The scripts interact with the virtual machine to install the required packages, and derive an input script (called the input descriptor) that will build the given sources. Gitian sets a defined hostname, username, uname (system information), build path, toolchain version and time. The input descriptor contains the version of the expected Ubuntu VM, the architectures to build on, the list of packages to install, a reference time that will be used to fake the time during the compilation to get rid of timestamp differences, the remote repositories to fetch, and a list of additional files to transfer into the VM. Finally, a custom script performs the compilation and takes care of the remaining sources of non-determinism. When the build is finished, the output additionally contains a list (the output descriptor) of the versions and hashes of all the dependencies involved during the compilation. Additional scripts allow for the submission of the resulting hashes signed, or verify these hashes against existing signatures.

Although many sources of non-determinism are taken care of by creating a clean environment that only contains the project to build and the downloaded dependencies, Gitian is difficult to generalize for other open-source software, and presents several limitations. In particular, it requires to be run on a Ubuntu OS to create a Ubuntu VM. The VM is run by qemu-kvm by default, a fork of the qemu virtualizer, that uses a kernel module (kvm) allowing kernel virtual machines. Such VMs rely on the hardware virtualization capabilities of the CPU (i.e., Intel VT-x and AMD-V). However, if one does not run Ubuntu on the physical machine, the outer Ubuntu OS needs to be run in a VM already, creating the need for a hypervisor with nested VM support for the compilation process. As nested hardware virtualization is not yet supported by processors, one must swap qemu-kvm for LXC containers, which can reduce the performance and increase the time required to compile a software package. More importantly, the compilation process must take place entirely in a Linux environment, requiring cross-compilation to target other platforms. Thus, Gitian cannot be used to compile TrueCrypt for Windows, as this application requires particularly non-replaceable compilers that run only on Windows.

4.2 Gitian for Tor

The Tor Browser Bundle (TBB) [36] compilation process builds upon Gitian, and provides a more automated process. TBB provides a browsing environment based on a modified version of Mozilla Firefox that automatically sends traffic through Tor. Various scripts, along with Gitian input descriptors, take care of the retrieval and authentication of the sources prior to compilation. Dependencies are downloaded through Tor itself, assuming Tor is already installed on the system (but not the Browser Bundle). This provides an additional layer of protection against a targeted attack on the source by providing anonymity to the downloading user (cf. threat #3). Dependencies are expected to match hashes that are embedded in the scripts. Then, dependency libraries are compiled and packaged in ZIP files, after which the modified Firefox is compiled and additional extensions are added to it for the final packages. These steps are divided into three distinct Gitian input descriptors.

Inside the compiling environment, various scripts cope with sources of non-determinism that are not considered by Gitian. For example, they allow the creation of deterministic ZIP and TAR archives that would normally include undesirable file timestamps. As we found, many of these scripts are quick fixes to make things work, but are not a perfect nor complete solution as of September 2014. In these scripts, one can notice many "FIXME" or comments such as "Crappy deterministic zip wrapper". TBB also must compile Firefox for all platforms (Windows, Linux and Mac OS) from a Ubuntu environment, involving challenging cross-compilation as previously mentioned. Finally, Gitian still sometimes produces non-deterministic output for unknown reasons [36].

4.3 Debian packages

Debian focuses on deterministic package build, as presented by Bobbio at DebConf'13 [4] and FOSDEM'14 [5]. Their approach is still experimental and is based on a special branch of the package management tools (dpkg). As of January 26, 2014, 67% of 6887 source packages were found reproducible. Their approach is however different than the one taken by Bitcoin and TBB. Also, instead of simply compiling an application, the focus is on Debian packages, which are the result of possibly more complex building steps including several tools. Instead of relying on resource-consuming VMs to perform the build, Debian adopts another philosophy: the problem of non-deterministic sources is treated at the root. For example, we noticed in our case study that one common source of non-determinism is the presence of embedded timestamps. Fortunately, Linux binaries (ELF format) do not contain embedded timestamps, contrary to Windows binaries (PE format). However in software packages, other tools in the toolchain may record timestamps. This is the case for `gzip`, `ar`, `tar`, `zip`, `jar`, and `javadoc`. The approach taken by Debian is then to patch these tools to add the option to get rid of timestamps in the tools themselves. Also, to prevent the project path to be included in binaries compiled by gcc, a special option can be passed to gcc, through the CFLAGS variable in the Makefile, that records a predefined build path in the binaries instead of the real one. However, this technique presents some incompatibilities, e.g., in build systems that reuse CFLAGS for other purposes that lead to non-deterministic output since CFLAGS contains the real path.

Bobbio [5] lists other sources of non-determinism that we did not encounter. He mentions about the file order of the `readdir()` function that is sensitive to the locale of the environment (whether it is set to UTF-8 or other languages). Another source difficult to identify depends "on an accident of filesystem layout at build time" [10]. A different file order listing can lead to different archives being created.

4.4 Other Linux distributions and software

Other Linux distributions have also started their own process towards reproducible/verifiable builds. Fedora proposes a few scripts to recompile source packages (SRPM) and compare them against the available builds [14]. The scripts support recursive verification that builds dependencies first. The comparison operates at a higher-level than just comparing the package hashes. Instead, packages are decomposed, and individual files are compared, according to an algorithm that takes care of the known semantic of the file structure. For example, ELF binaries are compared through their disassembled instructions. Thus, Fedora's approach is an example of builds that can be verified, however are not made by a deterministic process. OpenSUSE provides build-compare, a script to compare compiled software packages [35]. The compilation process is however not handled, and the comparison process also takes place at a higher-level than the simple file hash comparison. The first revision of these scripts dates back from January 2009. NixOS also attempts to achieve deterministic builds [12]. Reproducible Build Manager [48] is a tool that aims at reproducing software packages of multiple Linux distributions at once. It is based on compilation in a controlled environment. Finally, Mozilla also noted the importance of deterministic build and wishes to bring it to Firefox [13].

4.5 Summary of current efforts

Each of these different open-source projects takes a different path to tackle the problem of verifiability. While Bitcoin developers first created Gitian for their own purpose, which does not generalize well, the adaptation by Tor may be a step in the right direction. Debian, as an operating system, can afford changing its own tools for the right purpose. Such liberty may however not exist in heterogeneous environment sharing both open- and closed-source components. Fedora only focused on comparing two builds, with the underlying assumption that only predictable differences will be present. However, their approach remains naive as they do not provide a way to recompile a project for verification. As we observed in our study, it may not be always possible to automate the verification of non-deterministic builds. We note nonetheless that a framework for reconstructing the original environment may help the verification of current and past software. We did not find any work in this direction.

5 Related work

Related projects about deterministic and verifiable builds have been discussed in Section 4. We present below other attempts to analyze TrueCrypt, which is the core part of our case study.

The Ubuntu Privacy Remix Team primarily assessed TrueCrypt 7.0a for Linux in 2011, based on previously unpublished review of past versions 4.2a, 6.1a and 6.3a [46]. They created a tool, tcanalyzer,[4] which helps the study of TrueCrypt containers' headers. This tool was used to verify the correctness of both the official build and their own build. They report that they could not find mistakes or backdoors in the encryption or the header format. They further reviewed the cryptographic algorithms in TrueCrypt. They argue about the possibility of hiding a container's cryptographic keys inside the salt value in the TrueCrypt headers in Linux or Windows; or, inside an unused section of the headers in Windows. They advise to recompile the source rather than using the official binaries to prevent such attacks. It is worth noting that in the Linux version, the unused space of the headers is filled with encrypted zeros, which can be verified; however, it contains encrypted random data in the Windows version, which is impossible to distinguish from a backdoor version of the keys. Our analysis proves, at least for the Windows version, that the binaries do not differ from the source, and hence do not include such backdoors. The team then presented the discovery of a weakness in the keyfile algorithm, by which it is possible to manipulate any file so that it has no effect when added as a keyfile to encrypt a container.

Sogeti, a French information technology consulting company, reviewed TrueCrypt 6.0a for Windows, Linux and Mac OS X, as part of a first-level security certification for information technologies (CSPN) for the French government in 2008 [39]. CSPN is a security certification formalized by the Central Information Systems Security Division (DCSSI), a government entity under the authority of the General Secretary for French National Defence. This test is to be performed in 30 man-days, and is meant to provide a reference opinion about the security of an application (TrueCrypt here). Sogeti's analysis of TrueCrypt reported that the cryptographic algorithms were implemented correctly, and provided a positive opinion about the application in general, even mentioning that "the product [TrueCrypt] inspires confidence". However, several vulnerabilities were found, regarding a BIOS memory leakage of the password size in case of a system-encrypted partition, memory leakage of

[4] `https://www.privacy-cd.org/en/using-upr/download`

the password after the creation of a volume, memory leakage of keyfiles path, memory leakage of the XTS secondary key of a volume after a backup of its header, and a denial-of-service attack against the TrueCrypt driver. Several best practices are also suggested to avoid the identified issues.

In 2013, Amossys proceeded with the same test against TrueCrypt 7.1a for Windows only [2], building up on the previous analysis. They also conclude that the implementation of the main functionalities is correct. They however point out few vulnerabilities that remained unfixed since the previous CSPN test, including the BIOS memory leakage, memory leakage of the last created volume's password, and memory leakage of keyfiles path after a volume is dismounted.

The Open Crypto Audit Project[5] mandated iSEC Research Labs for a security assessment of selected security-sensitive parts of TrueCrypt 7.1a for Windows in 2014 [25]. iSEC identified 11 vulnerabilities, including two integer overflow vulnerabilities, possible leakage of sensitive information from the pagefile, various internal information leakage, and lack of security checks in the bootloader. They also propose corrections. However, no serious flaws were found. Adding our results about the past 16 versions helps to build trust around TrueCrypt.

6 Conclusion

As few users compile security-critical open-source software themselves from the source, there should be a way to guarantee that the official build is indeed compiled from the published source. This guarantee would prevent malicious/coerced authors from inserting backdoors in the compiled version only, and would also defend against targeted attacks. It can be offered thanks to a verifiable build that enables reproducing the official build. Verifiable build can be achieved either by a semantic comparison between the official and recompiled files, an approach taken by Fedora and openSUSE; or through a deterministic build process, which can be repeated and would always provide the same output, hence exactly matching the official build. However, sources of non-determinism can be difficult to isolate and reproduce. A perfectly deterministic build can only be achieved if all variables can be controlled. In our case study, we encountered a source of non-determinism based on the brand of the CPU of the building machine, which only showed up after we dug into several versions of the same application, and found no documentation about it. This problem leads to the following conclusion: it is not possible to ensure deterministic builds over time if the build process relies on closed-source software for which an exact documentation is unavailable. Also, through our analysis of 16 versions of TrueCrypt for Windows, we can conclude that verifying old software packages that inherently do not provide a guaranteed deterministic build, can turn into a forensic case in which one needs to gather all the appropriate tools that may have impacted the authors' build, and explain the reasons behind any oddities. In the end, we concluded that the binary files of TrueCrypt for Windows from version 5.0 to and 7.1a match the available source code. Our hope is that the challenges as uncovered through our TrueCrypt case study, and other concurrent projects, would eventually help guide designing future deterministic/verifiable build processes, which are critical for trusting security-critical software.

[5]`http://opencryptoaudit.org/`

7 Acknowledgments

We are grateful to anonymous ACSAC2014 reviewers for their suggestions. We also wish to thank the anonymous contributors for providing copies of old Microsoft development kits and pointing few additional details in our analysis. The second author is supported in part by an NSERC Discovery Grant and an FRQNT nouveau chercheur grant.

A Appendix

In this appendix, we list the MD5 and SHA1 hashes of the meaningful files involved in this project.

File:	TrueCrypt 5.0 Source.zip
MD5:	bfbd2616da3c3a35ff5c90e3e65df159
SHA1:	1d98f8e7130565bb73f525a8f4bba9b766315ad3
File:	TrueCrypt Setup 5.0.exe
MD5:	a3d94337991b4b84ead758d868408823
SHA1:	b0206174b69f2b471f7dcbe9a7b7075247cb0f24
File:	TrueCrypt 5.0a Source.zip
MD5:	6910802e7467329c0d87f381d2b901fc
SHA1:	5e1a88e101d601b77aa4c9f3451ab1535464664a
File:	TrueCrypt Setup 5.0a.exe
MD5:	4ec2b386f5d786b3017727aaecf28aa8
SHA1:	4ccb2a44ccbc978ca2042b9bc66833f14f13f6d4
File:	TrueCrypt 5.1 source.zip
MD5:	9d2198ce9b55a683d6d66166e796a1d0
SHA1:	9bd6abeb2911a612db02d82ecda4ba4a111989aa
File:	TrueCrypt Setup 5.1.exe
MD5:	df8385f648245ad4ba287089f5ea2b70
SHA1:	f55feb7da91c257d3ad3b0a4b01798b1ef06b89c
File:	TrueCrypt 5.1a Source.zip
MD5:	e969d5e7281c3f2fbc9cd075bb291441
SHA1:	6365b19105873f5bf4e86a6228c63bdf5524cdfa

File:	TrueCrypt Setup 5.1a.exe
MD5:	0b02b6a8b9437f8968cbe8719722079b
SHA1:	182f06a50a5baae19f9393b4c4632cd60f98debe
File:	TrueCrypt Setup 5.1a-2.exe
MD5:	9f2c390917d60aa2f729516cd1a6818f
SHA1:	0bbdccaa1a13c6027a44a3c5e449c2c9b2d36484
File:	TrueCrypt 6.0 Source.zip
MD5:	388698513e98afed053018b81cf9f371
SHA1:	fc81ea593c6555278c94b62a4e9ac8c73dd5069b
File:	TrueCrypt Setup 6.0.exe
MD5:	ec0827315825a035ff9a4203ddddfef7
SHA1:	3358dc3b94f8f0aa480846643026702f71f5c630
File:	TrueCrypt 6.0a Source.zip
MD5:	4b2f6e22f654bd6b93652926eea83290
SHA1:	b8f3c5c9112399001d9d51f36f1204910efbd124
File:	TrueCrypt Setup 6.0a.exe
MD5:	f5291fb74063ac4a57c069996623ea0b
SHA1:	21354e58584ebc97c8d96fcd7afda82b5070f116
File:	TrueCrypt 6.1 Source.zip
MD5:	105c51e624a1b96dff20ee56772ed956
SHA1:	268dab57a6eb9bb84c9fe22d9fb612a84cd7759a
File:	TrueCrypt Setup 6.1.exe
MD5:	2da0a448db5aa4e4770aecbf0357e008
SHA1:	b65d989e1276752b506367ab530fd551bb7ec699
File:	TrueCrypt 6.1a Source.zip
MD5:	4f00f836a644251d0d72b7ba32aaa0d5
SHA1:	72aeec5791fe46ffe9e7726451796e3e4f0d099c
File:	TrueCrypt Setup 6.1a.exe
MD5:	c413ecd820d2f912996ae86327b0d622
SHA1:	03a17dcfe5955f1316dffa2453e004e82ee2eed9
File:	TrueCrypt 6.2 Source.zip
MD5:	c55272f10b28122d5df8a61c29b84d11
SHA1:	f829a33b46a404b33981c799d530e50a7b5a2587
File:	TrueCrypt Setup 6.2.exe
MD5:	dc41720d117bd0e57288cec56d81ae8a
SHA1:	f836459553ed20174ed209cce1a0c700b1e36762
File:	TrueCrypt 6.2a Source.zip
MD5:	1fe56a53268484fa969f379d39939c5f
SHA1:	2032aa51fabb6a98ac2943e0845f3021242ba556
File:	TrueCrypt Setup 6.2a.exe
MD5:	75ceb941930f7900b6acf3d20944198e
SHA1:	dabb0e79b4ae7c45b17494b300396cc4868cb50b
File:	TrueCrypt 6.3 Source.zip
MD5:	60750ea11f6c08ef948f5c5aa1273267
SHA1:	aa3a52a738e20496ba47fce841883c886718bc4a
File:	TrueCrypt Setup 6.3.exe
MD5:	09894a801d343000a06649b5d5bebd4c
SHA1:	5918eee83832432fd51605ee7179964dbed29078
File:	TrueCrypt 6.3a Source.zip
MD5:	6c1f585957cb07e58c51732c83dad1e0
SHA1:	d21d22754584e419cda332d4e9561145d79d3475
File:	TrueCrypt Setup 6.3a.exe
MD5:	e14e7bd954482e5f43f9f8ce0ab2f7e2
SHA1:	2a31c146a5a4dbff00884678d8c2eca44928e03d
File:	TrueCrypt 7.0 Source.zip
MD5:	c8143751e0a8e681fead53e3855d025d
SHA1:	ee70ab85801d38f1ca1047bfdd7038c9d135de93
File:	TrueCrypt Setup 7.0.exe
MD5:	eadd4ae48541b830638f279d83938497
SHA1:	0be2bc7aa1431c4c10eedcf53a234b4959888111
File:	TrueCrypt 7.0a Source.zip
MD5:	752479c674bc18d6bcf55d056560f0a7
SHA1:	8f9bf2ae13461fb3bfb4d1f7acb76c7c1c7ed29d
File:	TrueCrypt Setup 7.0a.exe
MD5:	354e280c4bb56704e3925770f282588f
SHA1:	9ebe5de6130deae5d361306bf0add7a6789f6fbc
File:	TrueCrypt 7.1 Source.zip
MD5:	f4fc60d227bc2ed6641415fecc09b6dc
SHA1:	0f053bf5a463c5c48cdcc9b0cb8c9ed3d5aa2fb2
File:	TrueCrypt Setup 7.1.exe
MD5:	d4b8e358da8f382be1facf2f368a5fb3
SHA1:	5910a05bf671a385c2c5967171aa1c5509a3d3ee
File:	TrueCrypt 7.1a Source.zip
MD5:	3ca3617ab193af91e25685015dc5e560
SHA1:	4baa4660bf9369d6eeaeb63426768b74f77afdf2
File:	TrueCrypt Setup 7.1a.exe
MD5:	7a23ac83a0856c352025a6f7c9cc1526
SHA1:	7689d038c76bd1df695d295c026961e50e4a62ea

References

[1] AMD. Software optimization guide for AMD64 processors, Sept. 2005.

[2] Amossys. Rapport de certification DCSSI-CSPN-2013/09, Oct. 2013. `http://www.ssi.gouv.fr/IMG/cspn/anssi-cspn_2013-09fr.pdf`.

[3] Bitcoin project. Bitcoin Core, 2014. `https://bitcoin.org/en/download`.

[4] J. Bobbio. Reproducible builds for Debian. In *DebConf'13*, Vaumarcus, Switzerland, Aug. 2013.

[5] J. Bobbio. Byte-for-byte identical reproducible builds? In *FOSDEM'14*, Brussels, Belgium, Feb. 2014.

[6] Conifer Systems. Build determinism. Blog article (Oct. 17, 2008). `http://www.conifersystems.com/2008/10/17/build-determinism/`.

[7] Conifer Systems. Machine-independent builds. Blog article (Sept. 15, 2008). `http://www.conifersystems.com/2008/09/15/machine-independent-builds/`.

[8] X. de Carné de Carnavalet. How I compiled TrueCrypt 7.1a for Win32 and matched the official binaries. Blog article (Oct. 21, 2013). https://madiba.encs.concordia.ca/~x_decarn/truecrypt-binaries-analysis/.

[9] X. de Carné de Carnavalet and M. Mannan. Challenges and implications of verifiable builds for security-critical open-source software. In *ACSAC'14*, New Orleans, LA, USA, Dec. 2014.

[10] Debian Wiki. ReproducibleBuilds. Wiki article visited on May 21, 2014. https://wiki.debian.org/ReproducibleBuilds.

[11] Debian Wiki. SameKernel. Wiki article visited on May 21, 2014. https://wiki.debian.org/SameKernel.

[12] E. Egorochkin. Deterministic (bit-perfect) builds, June 2013. nix-dev mailing list. http://lists.science.uu.nl/pipermail/nix-dev/2013-June/011357.html.

[13] B. Eich. Trust but verify. Blog article (Jan. 11, 2014). https://brendaneich.com/2014/01/trust-but-verify/.

[14] Fedora Project. Reproducible builds for Fedora. https://github.com/kholia/ReproducibleBuilds.

[15] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *HotOS'97*, Cape Cod, MA, USA, May 1997.

[16] M. Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *NSPW'10*, Concord, MA, USA, Sept. 2010.

[17] R. Gallagher and G. Greenwald. How the NSA plans to infect 'millions' of computers with malware. News article (Mar. 12, 2014). https://firstlook.org/theintercept/article/2014/03/12/nsa-plans-infect-millions-computers-malware/.

[18] Gitian.org. Gitian: a secure software distribution method, Oct. 2009.

[19] GnuWin project. CoreUtils and gzip for Windows. http://sourceforge.net/projects/gnuwin32/.

[20] J. Gordon. The RSDS pdb format, 2010. http://www.godevtool.com/Other/pdb.htm.

[21] A. Gostev. 'Gadget' in the middle: Flame malware spreading vector identified. Blog article (June 4, 2012). https://securelist.com/blog/incidents/33081/gadget-in-the-middle-flame-malware-spreading-vector-identified-22/.

[22] Hex-Rays.com. Fast library identification and recognition technology, Feb. 2012. https://www.hex-rays.com/products/ida/tech/flirt/index.shtml.

[23] IEEE and The Open Group. dd. The Open Group Base Specifications Issue 7.

[24] Intel. Intel 64 and IA-32 architectures software developer's manual, Feb. 2014.

[25] iSEC. Open Crypto Audit Project - TrueCrypt - Security assessment, Apr. 2014.

[26] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and implementing malicious hardware. In *USENIX LEET'08*, San Francisco, CA, USA, Aug. 2008.

[27] H. Kirsch. The theory of build systems, Sept. 2013. http://www.pifpafpuf.de/BuildTheory.html.

[28] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May 2014.

[29] E. Lippert. Past performance is no guarantee of future results. Blog article (May 31, 2012). http://ericlippert.com/2012/05/31/past-performance-is-no-guarantee-of-future-results/.

[30] J. Menn. Exclusive: NSA infiltrated RSA security more deeply than thought - study. Reuters news article (Mar. 31, 2014). http://www.reuters.com/article/2014/03/31/us-usa-security-nsa-rsa-idUSBREA2U0TY20140331.

[31] Microsoft. Microsoft Portable Executable and Common Object File Format specification v8.3, 2013.

[32] Microsoft. Profile-guided optimizations, 2013. http://msdn.microsoft.com/en-us/library/vstudio/e7k32f4k.aspx.

[33] Microsoft. Specify symbol (.pdb) and source files in the Visual Studio Debugger, 2013. http://msdn.microsoft.com/en-us/library/ms241613.aspx.

[34] Mozilla Developer Network. Building with profile-guided optimization, Aug. 2013. `https://developer.mozilla.org/en/docs/Building_with_Profile-Guided_Optimization`.

[35] openSUSE Build Service. Build result compare script. `https://build.opensuse.org/package/show/openSUSE:Factory/build-compare`.

[36] M. Perry. Deterministic builds part one: Cyberwar and global compromise. Tor Project article (Aug. 20, 2013). `https://blog.torproject.org/blog/deterministic-builds-part-one-cyberwar-and-global-compromise`.

[37] D. Pistelli. Microsoft's Rich signature (undocumented). Blog article (Nov. 11, 2010). `http://www.ntcore.com/files/richsign.htm`.

[38] PrivacyLover.com. Analysis: Is there a backdoor in Truecrypt? Is Truecrypt a CIA honeypot? Blog article (Aug. 14, 2010). `http://www.privacylover.com/`.

[39] SOGETI Infrastructure Services. Rapport de certification DCSSI-CSPN-2008/03, Dec. 2008. `http://www.ssi.gouv.fr/IMG/cspn/dcssi-cspn_2008-03fr.pdf`.

[40] StackOverflow.com. How do I build TrueCrypt on Windows?, Nov. 2012. `http://stackoverflow.com/questions/13379644/how-do-i-build-truecrypt-on-windows/13414137#13414137`.

[41] The Netwide Assembler bug tracking system. NASM crashes when building x64 .obj file. Bug ticket (Oct. 6, 2009). `http://sourceforge.net/p/nasm/bugs/469/`.

[42] The PaX Team. Address space layout randomization, Mar. 2003. `http://pax.grsecurity.net/docs/aslr.txt`.

[43] K. Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, Aug. 1984.

[44] Tor Project bug track system. Improve software assurance. Bug ticket (May 31, 2012). `https://trac.torproject.org/projects/tor/ticket/6008`.

[45] TrueCrypt Foundation. TrueCrypt.

[46] Ubuntu Privacy Remix Team. Security analysis of TrueCrypt 7.0a with an attack on the keyfile algorithm. Technical report (Aug. 14, 2011). `https://www.privacy-cd.org/downloads/truecrypt_7.0a-analysis-en.pdf`.

[47] M. Uecker. Building packages three times in a row. Debian mailing list. `https://lists.debian.org/debian-devel/2007/09/msg00746.html`.

[48] N. Vigier. Reproducible Build Manager, 2014. `http://rbm.boklm.eu/`.

[49] J. Walton. An analysis of the Windows PE checksum algorithm, Mar. 2008. `http://www.codeproject.com/Articles/19326/An-Analysis-of-the-Windows-PE-Checksum-Algorithm`.

[50] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference*, Boston, MA, USA, June 2008.

[51] D. A. Wheeler. *Fully Countering Trusting Trust through Diverse Double-Compiling.* PhD thesis, George Mason University, Oct. 2009.