The Sad History of Random Bits

George Markowsky

School of Computing & Information Science, University of Maine,
markov@maine.edu

Received 5 February 2014; Accepted 27 April 2014;
Publication 2 June 2014

Abstract

In this paper we examine the history of using random numbers in computer programs. Unfortunately, this history is sad because it is replete with disasters ranging from one of the first pseudo-random number generators, RANDU, being very bad to the most recent efforts by the NSA to undermine the pseudo-random number generator in RSA's BSAFE cryptographic library. Failures in this area have been both intentional and unintentional, but unfortunately the same sorts of mistakes are repeated. The repeated failures in getting our "random numbers" correct suggests that there might be some systemic reasons for these failures. In this paper we review some of these failures in more detail, and the 2006 Debian OpenSSL Debacle in great detail. This last event left users of Debian and its derivatives with seriously compromised cryptographic capabilities for two years. We also illustrate how this failure can be exploited in an attack. We also modify the concept of a system accident developed in the work of Charles Perrow [1]. We identify some system failures in building pseudo-random number generators and offer some suggestions to help develop PRNGs and other code more securely.

Keywords: Debian; system accident; SSL; SSH, Bitcoin, cryptography; security breach; software engineering, PRNG, pseudo-random numbers, booby trap, BSAFE, Dual_EC_DRNG.

Journal of Cyber Security, Vol. 3 No. 1 , 1–26.
doi: 10.13052/jcsm2245-1439.311
© 2014 *River Publishers. All rights reserved.*

1 Introduction

In most cases, when something goes wrong, we look for someone to blame. The underlying assumption is that someone must have been the direct cause of each particular mishap. In [1], Charles Perrow introduced the concept of a *system accident*, something he also called a *normal accident*. The basic idea is that while a serious accident might include a number of smaller events the serious accident occurred because the system allowed the smaller events to interact in a manner that combined their contributions. While his analysis targets physical systems, his emphasis on looking at the various components of a system is valuable for virtual systems as well. In this paper, we develop a modified version of a system accident that we think is easier to apply to software systems. We will apply it to a number of incidents involving pseudo-random numbers generators and examine it in detail in the analysis of a little known incident that occurred in 2006 in which the Debian SSL library was compromised. It took two years before someone noticed that the library had been compromised during which time SSL and other services that use the SSL libraries, like SSH, were vulnerable. We conclude with some suggestions on improving the system. This article is a much expanded version of [2].

2 Randomness

People in general are not comfortable with the concept of randomness. People like to feel that things happen for a reason. Sentiments such as these date back to the earliest writings and are found in the theological writings of all people. It is widely stated and believed that "God has a plan for everyone" and things happen for a reason. These ideas were incorporated into modern science as the following quotations illustrate.

> We may regard the present state of the universe as the effect of its past and the cause of its future. An intellect which at a certain moment would know all forces that set nature in motion, and all positions of all items of which nature is composed, if this intellect were also vast enough to submit these data to analysis, it would embrace in a single formula the movements of the greatest bodies of the universe and those of the tiniest atom; for such an intellect nothing would be uncertain and the future just like the past would be present before its eyes.
>
> — Pierre Simon Laplace, *A Philosophical Essay on Probabilities*

> As I have said so many times, God doesn't play dice with the world.
>
> — Albert Einstein

Not surprisingly, these sentiments have found themselves into our literature.

> What object is served by this circle of misery and violence and fear? It must tend to some end, or else our universe is ruled by chance, which is unthinkable.
>
> — Sherlock Holmes, The Cardboard Box

Fortunately, for the human race some people decided to explore randomness and chance and see if there might be some "laws" or principles that govern it. Some of the earliest efforts to understand chance and randomness were inspired by gamblers. Girolamo Cardan (1501–1576) studied the questions associated with games of chance at the behest of gamblers. Other early contributors to the theory of probability were Fermat, Pascal and Descartes.

Another early effort to understand chance in the affairs of humans came when John Graunt (1620–1674) published *Natural and Political Observations Made Upon The Bills of Mortality* in London in 1662. He made the very surprising observation that while he could not tell who would die in a given year and how they would die, he could very accurately predict how many people would die in a "normal" year, and even how many people would die from what cause.

These observations from gambling and population studies fueled the development of probability and statistics and gave people some confidence that randomness followed some laws. The science of thermodynamics contributed significantly to the development of probability and statistics.

The birth of quantum mechanics in the twentieth century made randomness a central feature of science and caused Einstein to utter the famous quote presented earlier in this paper. Despite all efforts to remove randomness as a central feature of quantum mechanics, randomness appears to be at the center of the physical universe and to some extent, the Universe is ruled by chance.

If our understanding of quantum mechanics and of other physical theories is correct, then we should be able to get truly random numbers from a number of physical processes. Knuth [3, p. 3] describes some devices that were used to physically generate random numbers in the early days of computing. There are a number recent systems that generate random numbers based on physical phenomena. For example, Fourmilab [4] provides random bits

derived from the radioactive decay of Cesium-137. Random.org [5] provides random numbers based on atmospheric phenomena. A joint effort between the Physics Department at Humboldt University (Germany) and PicoQuant GmBH [6] has produced the PQRNG 150, a quantum mechanical random number generating device that can be purchased [7]. One of the problems with standard physical devices is that they produce random bits very slowly. PicoQuant claims that the PQRNG 150 can produce more than 150 Mbits per second of random bits. Another option that might help to produce large quantities of truly random data is described in a paper by Gallego, Masasnes, De La Torre, Dhara, Aolita and Acfin [8].

3 Pseudo-Random Number Generators

From the very beginning of computing, there was a need for random numbers. Knuth [3, Chap. 3] provides a short history of the use of random numbers in computing and a lot of technical information about random numbers that should be more widely known. In 1946 John von Neumann first suggested using computer programs to generate "random numbers" [3, p. 3]. He also made the following statement about such programs.

> Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.
>
> — John von Neumann (1951) [3, p. 1]

As was noted from the beginning of modern computation, "random numbers" generated by some sort of deterministic computation cannot be truly random. They can, however, appear to be random. Such apparently random number sequences are referred to as *pseudo-random numbers* and the programs that generate them are known as *pseudo-random number generators* or *PRNGs*. To be truly useful, pseudo-random number sequences need to pass a wide variety of statistical tests. Knuth [3, Chap. 3] provides a good discussion of these tests and how they can be implemented.

As history has shown and this paper will document, people have made and continue to make serious mistakes in designing and running PRNGs. This inspired Robert Coveyou to write a paper in 1970 entitled "Random Number Generation Is Too Important to Be Left to Chance" [9]. In a similar vein, Knuth [3, p. 6] states that "*...random numbers should not be generated with a method chosen at random.*

We will now give a very general description of how PRNGs work. Generally, there is a finite set, S, of some sort. S is often a set of integers or real numbers. The PRNG can be thought of as a (deterministic) function $f : S \rightarrow S$. In general, S should be a large set and f should be a function that is easy to calculate, but whose inverse should be difficult to calculate. We start at some value a in S, called the *seed* and generate the sequence $a, f(a), f(f(a)), f(f(f(a))), ...$ If we have done our work well, the sequence will appear to be random and pass whatever statistical tests we can devise.

Figure 1 shows the general structure of a PRNG. To understand this figure note that the sequence $a, f(a), f^2(a), f^3(a), ...,$ where $f^2(a) = f(f(a)), f^3(a) = f(f(f(a)))$, etc must repeat since S is finite. Suppose i is the smallest integer such that $f^i(a) = f^j(a)$ for some $i < j$. Note that the sequence $a, f(a), ..., f^i(a)$ consists of distinct values. The value i is called the *index* of f. Pick the smallest value for j such that $i < j$ and $f^i(a) = f^j(a)$. The value $j - i$ is called the *period* of f and we will denote it by p. Note that $f^q(a) = f^{q+tp}(a)$ for all $q \geq i$ and all integers t. Generally, the function f is supplemented by a projection function π which converts the values $f^q(a)$ into whatever form is desired. For example, we might want to generate a pseudo-random sequences of 0s and 1s and π would convert the values $f^q(a)$ into 0s and 1s. Note that Figure 1 would remain the same regardless of whether

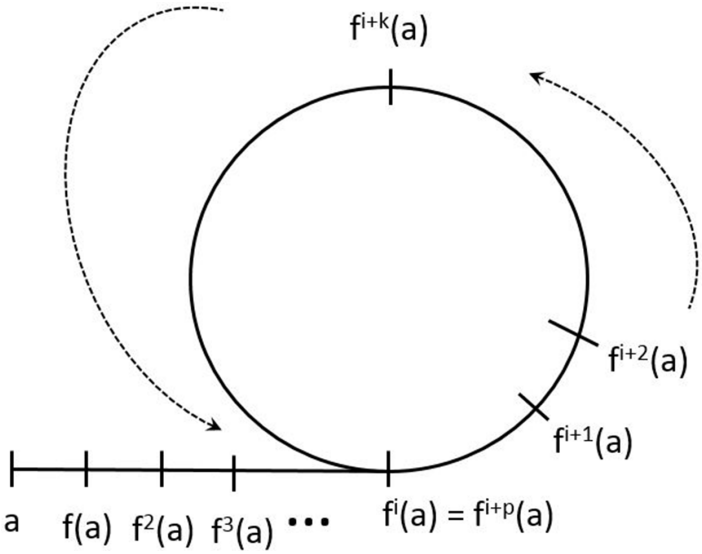

Figure 1 The Structure of a PRNG

f is an injective function or not. For some bounds on index and period, see Markowsky [10].

In general, the index is of little interest to PRNG designers and is often 0. Of great interest is the value of the period. In general, values for the PRNG are constrained to values that appear on the cycle. If the period is small, say less than 100,000, the PRNG is not very good for security purposes because an attacker can use a brute-force approach to compromising security. Just as 2 character passwords are not very secure because an attacker can try all of them, PRNGs with small periods are not very secure.

According to Wikipedia [11], the Mersenne Twister developed by Makoto Matsumoto and Takuji Nishimura [12] is currently the most used PRNG. It is based on the Mersenne prime $2^{19937} - 1$ and has a period of $2^{19937} - 1$. It passes most, but not all, randomness tests, but in its native form is not good for cybersecurity purposes in part because knowing 624 consecutive values permits one to figure out exactly where the PRNG is on its cycle and to generate all future outputs.

Besides Knuth's volume [3] which we have mentioned several times, we might also mention the article by Jerry Dwyer [13] that gives a more concise overview of PRNGs along with code examples. Wikipedia also provides some useful and current information about PRNGs.

4 RANDU

RANDU [14] is a PRNG that was introduced in the 1960s and used extensively for more than a decade. RANDU fails most randomness tests and is a bad PRNG. Knuth [3, p. 107] feels very strongly about its use as can be seen in the following quote where line 12 refers to a table containing the test results of RANDU and other PRNGs.

> Line 12 was, alas, the generator actually used on such machines in most of the world's scientific computing centers for more than a decade; its very name RANDU is enough to bring dismay into the eyes and stomachs of many computer scientists! ... the generator fails most three-dimensional criteria for randomness, and it should never have been used.

RANDU is an important example for us of a system failure. It was used by many people unfamiliar with the theory of PRNGs based on trust that a PRNG would not be used without careful thought and reasoning. It was used

in all sorts of simulations and designs including those of nuclear weapons and reactors. One only hopes that RANDU's weaknesses did not lead to serious problems.

5 The Debian SSL Debacle

The discussion of the Debian SSL Debacle in this paper is based on papers by Ahmad [15], blog entries by Cox [16] and Schneier [17], the video by Bello and Bertacchini [18], and the video by Applebaum, Zovi and Nohl [19].

Ironically, the Debian SSL Debacle was caused by Debian developers trying to do a good thing. As is well known, memory errors are a fertile breeding ground for software failures of all types and often lead to breaches in cybersecurity. The Open Source community has developed tools to help find memory errors in software. In particular, "Valgrind" [20] is one such tool.

One of the developers running Valgrind on the Debian source code received a message warning about the use of an uninitialized variable in the function MD_Update in two places. Normally, this is an error in programming, but in this case the uninitialized variable was used together with other components to increase the amount of randomness in the OpenSSL module of the Debian operating system. In one location in the code, the use of MD_Update brought in other critically important sources of randomness in addition to the randomness imported from the uninitialized variable. Removing this instance of the MD_Update function critically damaged the OpenSSL module.

The OpenSSL module is used to encrypt communication for the operating system. For example, a person signing into a "secure" website is typically dependent on the OpenSSL module for providing good cryptographic strength. In general, the more randomness in a cryptographic system the better, so reducing the amount of randomness is a serious error.

The developer discussed with other Debian developers and also corresponded with OpenSSL developers. We shall examine the conversation in more detail in Section 10. He received an ambiguous reply that he interpreted as an approval to remove the lines in general, so he removed them from the program by commenting them out. It may be that the reply was just approving the removal of the lines in question just for debugging purposes, but the reply is poorly constructed and it is easy to see how the reply might have been interpreted as a blanket approval to remove those lines.

The main consequence of this action was to limit the PRNG so it would produce only 32,768 distinct values. While this is a large number for manual efforts, it is a relatively small number for a computerized "brute-force attack."

The lines were commented out in 2006, and it was not until 2008 that the error was discovered and the weakness of the resulting cryptographic system was established. Thus for about two years the cryptographic capabilities of Debian Linux were severely compromised. Hence the title of Ahmad's paper "Two Years of Broken Crypto" [15].

6 PRNG Problems in Android

Ironically, after the 2006 Debian SSL Debacle, Google committed the same sort of error in its PRNG for its Android operating system which is the most widely used operating system for mobile phones. The error again was not using all proper uses of entropy for the seed. The details are given in the following post to Google's Android Developers Blog [21] which appeared on August 14, 2013.

> The Android security team has been investigating the root cause of the compromise of a bitcoin transaction that led to the update of multiple Bitcoin applications on August 11.
>
> We have now determined that applications which use the Java Cryptography Architecture (JCA) for key generation, signing, or random number generation may not receive cryptographically strong values on Android devices due to improper initialization of the underlying PRNG. Applications that directly invoke the system-provided OpenSSL PRNG without explicit initialization on Android are also affected. Applications that establish TLS/SSL connections using the HttpClient and java.net classes are not affected as those classes do seed the OpenSSL PRNG with values from /dev/urandom.
>
> Developers who use JCA for key generation, signing or random number generation should update their applications to explicitly initialize the PRNG with entropy from /dev/urandom or /dev/random. A suggested implementation is provided at the end of this blog post. Also, developers should evaluate whether to regenerate cryptographic keys or other random values previously generated using JCA APIs such as SecureRandom, KeyGenerator, KeyPairGenerator, KeyAgreement, and Signature.
>
> In addition to this developer recommendation, Android has developed patches that ensure that Androids OpenSSL PRNG is initialized correctly. Those patches have been provided to OHA partners.

We would like to thank Soo Hyeon Kim, Daewan Han of ETRI and Dong Hoon Lee of Korea University who notified Google about the improper initialization of OpenSSL PRNG.

7 The Bitcoin Compromise

As noted in Section 6, the Android PRNG was supposed to get a "random" seed from /dev/urandom, a protected system root file. However, the programmers did not reference this file in the code and left it to the user to pick a "random" seed. Of course, most users were not aware that they needed to do anything with the predictable result that a "random" seed was not picked in all cases.

So far there have been no verified consequences of the Debian OpenSSL Debacle, but we have a documented instance of some serious consequences of the Android failure. In particular, Goodin [22] describes the theft of $5,700 in bitcoins as a result of this flaw. It should be noted that the massive theft of $100 million in bitcoins [23] appears to be unrelated to this flaw and seems to be an instance of old fashioned, low tech fraud.

8 RSA, BSAFE and the NSA

The famous computer security firm RSA offers a cryptography library certified by NIST called BSAFE [24]. BSAFE can use a variety of PRNGs, but from 2004 to 2013 it made Dual_EC_DRBG the default PRNG for its library. This was a curious choice because as noted by Matthew Green [25]:

> Not only is Dual_EC hilariously slow – which has real performance implications – it was shown to be a just plain bad random number generator all the way back in 2006.

In particular, Microsoft Researchers Dan Shumow and Niels Ferguson at the Crypto 2007 rump session demonstrated that Dual_EC_DRBG might be compromised by the existence of a backdoor in this PRNG [26]. Shumow and Ferguson did not claim that a deliberate backdoor was placed in the PRNG, but warned that there could be one. Despite warnings from cryptographers about the dangers of using this PRNG, RSA continued to have use it as the default PRNG for BSAFE.

The Snowden leaks [24, 27, 28] showed that the concern about a backdoor were valid and that indeed such a backdoor was inserted into the PRNG by the NSA. In December 2013 Joseph Menn [29] revealed that the NSA

paid RSA $10 million to make Dual EC DRBG the default PRNG for BSAFE.

This incident provides an example of the deliberate sabotage of a PRNG to obtain information. RSA is now left with the task of dealing with the fallout from this event. One embarrassing event was RSA warning its own customers about using the defaults in its own package [30]. Another embarrassing event was the withdrawal of a number of prominent cryptography experts from the RSA Annual Conference [31].

9 System Accidents

Perrow's book [1] contains detailed accounts of many industrial accidents as well as accidents in various other areas. Among the most fascinating accounts are those involving ships colliding in the ocean. A number of these collisions start off with the ships on courses that would not lead to a collision. Unfortunately, actions taken by the crew cause the collision. An example of such a collision is shown in Figure 2 which is derived from [1, p. 210]. Clearly, a collision is something both crews were eager to avoid, yet their actions led to a collision. Clearly, there is something about the marine transportation system that contributed to the collisions. There may, indeed, be failings on the part of the crew, but there are system failures that contributed to the accident.

Perrow uses the term system accident or normal accident throughout his book [1] but nowhere does he give a clear and concise definition. Furthermore, his focus is on physical systems and some of the factors he focuses on are less

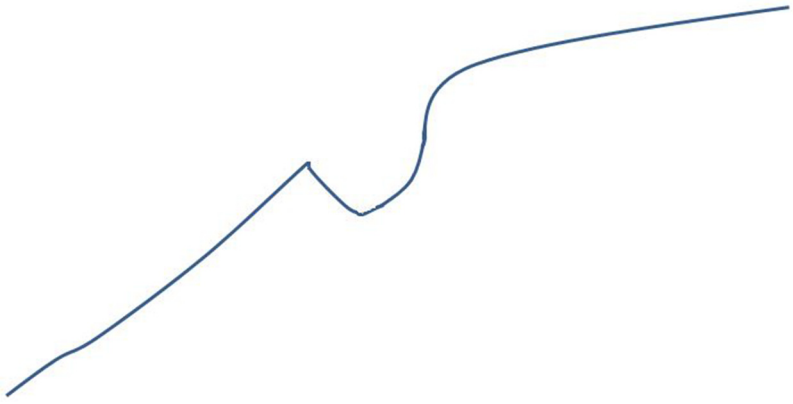

Figure 2 A Non-Collision Course Collision

relevant to virtual systems. We will review the appropriate terms from [1] and then propose how to interpret them for virtual systems.

First, Perrow spends a bit of time distinguishing between "incidents" and "accidents." Both of these "involve damage to a defined system that disrupts the ongoing or future output of that system" [1, p. 64]. Furthermore, Perrow divides systems into four levels: part, unit, subsystem and system. An incident is an event that occurs at the first two levels and an accident is an event that occurs at the last two levels. This leads Perrow to the follow formal definition [1, p. 66]. Note that the acronym ESF stands for engineered safety feature.

> We are ready for a formal definition. An *accident* is a failure in a subsystem, or the system as a whole, that damages more than one unit and in doing so disrupts the ongoing or future output of the system. An *incident* involves damage that is limited to parts or a unit, whether the failure disrupts the system or not. By disrupt we mean the output ceases or decreases to the extent that prompt repairs will be required. Since we have drawn a dividing line between the unit and the subsystem, and since many of the ESFs are clustered around that dividing line, it will often mean that an ESF will be one of the components that fails.

There are many features of the preceding definition that do not apply to the Debian OpenSSL Debacle. For example, the error that was introduced never interfered with the operation of the Debian operating system. In fact, the operating system ran without incident for two years before anyone noticed the error. In addition, it is not clear who was damaged by this bug and to what extent. We will address this issue further in Section 11. Clearly, the potential was there for massive cybersecurity breaches but we have no easy way to tell which security breaches were the result of this failure. In the case of physical systems and accidents like factory explosions and ship collisions it is often obvious what is damaged and to what extent. Of course, software systems can also cease to function as a result of errors or attacks, but it is not unusual for large software systems to have many errors and to function adequately. Of course, many of the errors are often minor and occur in very limited circumstances.

One of the major points that Perrow makes in [1] is that people, in particular system owners, like to find a scapegoat so that they are not obligated to fix the system. Often this is because system owners believe that it is cheaper to deal with the occasional mishap rather than fix the system [1, p. 67]. Perrow also

has some very interesting points to make about the complexity of systems and the tight coupling of systems. There is no doubt that many software systems such as operating systems are extremely complex. Furthermore, there are few complex physical systems that do not depend on a complex software system for control.

In 2008 Perrow wrote an unpublished paper [32] dealing with software failures. While his analysis of technical issues leaves much to be desired, this paper is a fascinating collection of software failures of varying severity and is worth reading. The distinction between incident and accident is a valuable one even for virtual systems. We will use the term *incident* to describe a failure that has caused or is likely to be more of an inconvenience or nuisance than a serious failure. We will reserve the term *accident* for a failure that has caused or has the potential to cause serious damage or loss.

Perrow believes that serious accidents are often not just the result of one person's mistakes. Rather, they are often the result of a sequence of minor mistakes which combine to produce the serious accident. In this, they are aided by features of the relevant system that make the accident more likely. For example, in [1] Perrow shows how the operator is often blamed even though various safety devices did not work properly or gave misleading information. He spends lot of time in his analysis of the Three Mile Island nuclear accident [1, Chap. 1] showing how the operators were unable to get correct information from some of the gauges, yet they were blamed entirely for the accident.

We want to make the concept of a *booby trap* central to our definition of a system accident. A booby trap is designed to hurt someone who gets in its way. Booby traps can be set deliberately or unintentionally. Booby traps are often set by property owners to protect their property. Many polities have laws or regulations against the setting of booby traps because the final result of a booby trap is unpredictable and can be much worse than the consequence of the crime the booby trap is designed to prevent. Many people realize that booby traps such as wiring shotguns to doors are not intelligent things to construct. In fact, it is not unusual for property owners to be killed by their own booby traps [33].

For software systems we define a *booby trap* as some feature that makes it more likely that a user will make an error. One famous example of a software booby trap is the famous loss of the Mars Climate Orbiter spacecraft in 1999 [34]. The problem was that the two engineering teams that worked on the software for this system used different systems of measurement. In particular, one team used the metric system and the other team used the imperial system for measurements. As one would expect, at some point someone forgot to

make the proper conversion and as a result a $125 million satellite was lost. It is clear that having two teams working with two different measurement systems is a booby trap. While one can try to blame "operator error" for the accident, it was clearly a system accident because the system was set up to make such an accident extremely likely. Despite this costly error, NASA continues to use both measurement systems for projects, although in 2007 it made the commitment to use the metric system for all operations on the lunar surface [35].

We propose the following definition: a software failure will be considered a *system accident* if it has serious consequences or the potential of having serious consequences and was caused by one or more booby traps. Our primary purpose is to identify booby traps in the software creation system, because even if they lead to minor errors in one situation, they can lead to more serious consequences in other situations.

10 The Debian SSL Debacle Revisited

In this section we will revisit the Debian OpenSSL Debacle from the point of view of finding booby traps in the open source development system that might be present in other systems as well. For convenient reference we will use DOD to refer to the Debian OpenSSL Debacle.

The first point to consider is the common lack of proper commenting in code. Many programmers either do not know how, don't have the time, or are too lazy write proper comments that really explain their reasoning and what the code does. They also tend not to point out booby traps in their code. This problem affects all types of software both proprietary and open source. This is *Booby Trap 1*. This booby trap was sprung in the DOD as illustrated by the fact that the developers who were fixing the "bug" did not properly understand the nature of the code they were working on.

Closely related to Booby Trap 1, is the fact that the programmers making the fix did not really understand the nature of what they were doing. To some extent this is *Booby Trap 2*, the lack of proper education. We view this as a system failure because the type of material found in Knuth [3, Chap. 3] is not generally taught in the computer science curriculum. Most computer science students have to take some sort of coursework in probability and statistics, but these courses typically do not talk about the problem of generating random numbers.

Closely related to the preceding booby traps, is the writing of overly clever code. This is code which does something in a very efficient and elegant manner

and often combines multiple operations into one. Such code needs a lot of commenting and is often hard to understand for other programmers. It is often hard to understand for the programmer who wrote it once the creative insight passes. We call this *Booby Trap 3* and it was sprung in the DOD as discussed below.

While being proactive and finding errors in software before they manifest themselves is a good thing, there is a fundamental danger in using automated tools because these tools do not understand the logic of programs and perform their analysis at a very low level. Such tools can report errors, which are not truly errors. This is *Booby Trap 4*, and it was sprung in the DOD as seen from in the correspondence between Debian developers [36].

Valgrind has the property that once it finds an "error" it will track the effects of that error as they propagate through the entire software system. On the one hand this seems like a good idea, but in practice it leads to a large number of error messages. Experience shows that providing humans with too many warnings tends to make them ignore the warnings or even shut the system down. Perrow [1] describes many instances where safety systems were shut down before accidents because they were overwhelming the operators. This is *Booby Trap 5*. It was sprung in the DOD as illustrated in [36].

There is another booby trap associated with the use of powerful software testing tools. This is the potential to lead developers to venture into parts of the code that they do not understand well. This is *Booby Trap 6*. It was sprung in the DOD as illustrated in [36]. In this correspondence one of the developers notes that two lines in particular are the cause of the Valgrind error messages. He also notes that the function of these lines is to add "uninitialized numbers to the pool to create random numbers." Unfortunately, this developer did not fully understand that other sources of randomness were involved as well. The rest of the discussion in [36] turned into a technical discussion about using Valgrind, and the key point about contributing entropy was buried in the other discussion. One of the developers in the discussion was not completely comfortable with the discussion and realized that he was out of his depth in accessing the entropy issue and he contacted the OpenSSL developers [37]. In his note he made the statements shown in Figure 3.

There is another key point that needs to be made. The Debian developers identified two lines of code involved in generating the multitude of Valgrind errors. Both lines introduced an indeterminate amount of entropy from the uninitialized memory location, but one of the lines also introduced entropy from other sources such as system time, the PID, the UID and the random number generator. By focusing on the uninitialized variable, the developers

overlooked all the other sources of randomness. Had these sources of entropy been introduced in separate lines, only the entropy coming from the uninitialized memory location would have been eliminated from the program. This is how Booby Trap 3 was sprung. The developers were led into this trap by inadequate commenting of the code which itself was too clever.

Figure 4 shows part of the reply 4 to the letter shown in Figure 3 is reproduced in Figure . It includes just one paragraph from the original letter which is emphasized. There are several problems with the reply. First, the reply begins with the phrase "Not much." It is not clear what "not much" refers to. If it refers to the sentence just above, then it suggests that the uninitialized variables do not add much entropy to the random number generator (RNG). This would support commenting out the lines in question. On the other hand, the original correspondence concluded with the line "What do you people think about removing those 2 lines of code?" It is possible that the writer was responding to that question and was stating that he did not think much of removing the two lines of code. Another ambiguity in the reply is the statement "If it helps with debugging, I'm in favor of removing them." Did the author mean that he was in favor of removing the lines but only for debugging purposes, or was he supporting the removal of the lines permanently? In any event, the Debian developer decided that this was the approval that he needed to remove the lines of code. The ambiguity in this communication was *Booby Trap 7*.

> What I currently see as best option is to actually comment out those 2 lines of code. But I have no idea what effect this really has on the RNG. The only effect I see is that the pool might receive less entropy. But on the other hand, I'm not even sure how much entropy some unitialised [sic] data has.
>
> What do you people think about removing those 2 lines of code?

Figure 3 Portion of a Letter to the OpenSSL Project

> *What I currently see as best option is to actually comment out those 2 lines of code. But I have no idea what effect this really has on the RNG. The only effect I see is that the pool might receive less entropy. But on the other hand, I'm not even sure how much entropy some unitialised [sic] data has.*
>
> Not much. If it helps with debugging, I'm in favor of removing them. (However the last time I checked, valgrind reported thousands of bogus error messages. Has that situation gotten better?)

Figure 4 A Reply to the Letter from Figure 3

11 Discovery, Mitigation and Consequences

Debian released Debian Security Advisory DSA-1571 [39] in 2008. It stated that Lucianon Bello discovered the flaw in the OpenSLL package used by Debian. It suggested some remediation such as regenerating various security keys. The announcement should have been enhanced to impress upon people the importance of this error. A variety of exploits aimed at this bug can be found in [18, 19, 39 – 42]. The sources just cited point to sources that should be consulted for additional exploits. We discuss an exploit in more detail in Section 12

Compromises affected not only Debian systems with the faulty software, but other systems that engaged in certain types of interactions with compromised computers. Reference [43] contains a discussion of weak keys, but this discussion would be very difficult for a non-expert to follow. Similarly, exploring DSA-1571 [39] provides many details, but again these would be difficult to follow for the non-expert. The failure to completely explain all consequences of this vulnerability and the failure to more widely alert the user community is *Booby Trap 8*.

There was another booby trap set by the Debian organization in the way that they handled the announcement of the vulnerability. In particular, they posted the vulnerability patch on May 7, 2008 but withheld the public announcement of the vulnerability until May 13, 2008. This is *Booby Trap 9*. The problem here is that there are skilled people who read code changes and who would understand the significance of this error even without the announcement. Not seeing the announcement at the same time as the patch, they would realize that there would be a window of opportunity to brute-force attack systems. Reference [19] reports that there was a sharp increase in the number of brute-force attacks against many hosts during the period between May 7 and May 13.

The inappropriate commenting out of two lines reduced the key space to a maximum of 2^{15} = 32,768 keys. The total number of keys is greater because the set of 32,768 depends on the system used. In any event, generating the total number of keys for all systems is feasible with limited equipment. [19] even demonstrates how to use Amazon Cloud Services to generate the keys in a relative hurry for under $25.

References [15 – 19, 42] all discuss the consequences of this failure. Of particular interest is the graph in Figure 5 which is derived from [42]. It appears that after nearly six months more than 40% of the vulnerable certificates had not been updated. This is *Booby Trap 10*.

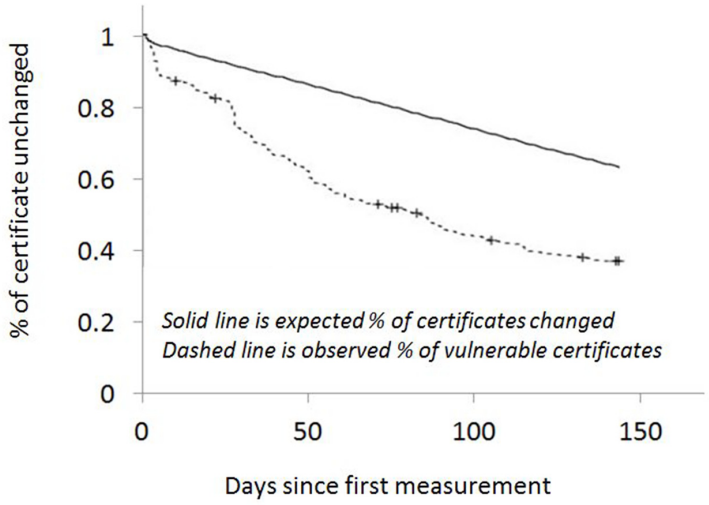

Figure 5 The Rate of Updating Vulnerable Certificates

12 An SSH Attack

In this section we wish to briefly describe in detail a possible attack against OpenSSH because of the weakness in OpenSSL. The reason OpenSSH was affected was because it uses the cryptography library in OpenSSL and hence uses the faulty PRNG from OpenSSL.

SSH supports a method for connecting to a server without using a password. The idea is based on the application of public key cryptography. In public key cryptography there are two keys that act as inverses of each other. Anything encoded using one key can be decoded using the other key. One key is called the *public key* and may be distributed without restriction. It can be published, displayed on a website or otherwise restricted. The other key is called the *private key* and must be maintained securely by the person wishing to receive messages encoded by the public key. In a successful public key system it should be extremely difficult (computationally impossible) to construct the private key given the public key.

Generally, the public and private keys are generated at the same time and are often based on a value supplied by a PRNG. If the PRNG is compromised and produces a small number of outputs, this would mean that only a small number of possible key pairs can be generated using that PRNG. Recall, that because of the reduced entropy to the PRNG in the Debian OpenSSL package, only 32,768 (2^{15}) key pairs are possible.

The passwordless login using SSH works as follows The person who wants to use such a feature generates a public-private key pair. The public key is stored on the server in a particular folder in the account space that belongs to the account which is to support passwordless login. Login then proceeds as follows. The user contacts the server and presents the login and the public key to the server. The server then checks whether the public key is present in the appropriate space in the account space and sends back a message encoded with the public key. In principle, only the holder of the matching private key can properly decode the message. Since the contents of the message are used as the basis of the link, the server has some confidence that the correct person has been connected.

With the Debian OpenSSL flaw only 32,768 key pairs were possible. This number of key pairs can be generated relatively easily to give all possible key pairs for key lengths of 1,024, 2,048 and 4,096 bits. The exploit proceeds as follows. The attacker picks a user ID of someone likely to use passwordless login such as a system administrator. Passwordless login is used by many system administrators because they want to quickly log into many different computers. The attacker then presents the user ID along with each of the public keys in turn until the server gives a proper response at which time the attacker uses the corresponding private key to initiate the session. While 32,768 choices seems like a lot, this is a not a large number of trials for a computerized brute force attack.

TJ O'Connor's book, *Violent Python* [45, pp. 41–55] describes such an attack in detail and provides Python code for this example. Details on SSH are available in the book by Barrett, Silverman and Byrnes [46]. A detector of weak key material is available at the Debian.org site [39] and directly at [47]. This detector is written in Perl and contains information about the keys.

The author would like to thank his RIT colleagues Bruce Hartpence and Bill Stackpole for some useful discussions about SSH.

13 Conclusions and Suggestions for Improvement

In this paper we identified ten booby traps that led to the DOD and contributed to worsening the consequences. We list them below with a brief description and some suggestions for dealing with each booby trap. It is clear that these booby traps are not unique to the DOD and perhaps the lessons learned here can be helpful elsewhere.

1. *Poor Commenting.* This is a standard problem in coding and it is not clear that we are making progress. Figure 6 [44] shows a more recent version of the code that was involved in the DOD. Note that while there is a comment warning people not to remove a particular instance of MD_Update, there is no explanation of why an uninitialized variable is being used. Furthermore, there are several other references to MD_Update in the same code section but there is no explanation of what these other calls are achieving. Poor commenting is a system failure. It is not properly taught in most computer science programs and there are few good tools for writing useful comments. Knuth [48, 49] has put forward a philosophy of programming and produced tools to support his approach, but so far this effort has had limited impact. More effort needs to be put into this initiative and similar initiatives. On a related note, more widespread use of *Test-Driven Development* [50, 51] can provide programmers with useful information about what code is supposed to do along with tests that can help limit errors stemming from code modification.
2. *Lack of Proper Instruction.* As noted in this paper, most computer science professionals have an inadequate basis for understanding the difficulties of using PRNGs. We suggest that material dealing with random numbers and PRNGs become part of the standard computer science curriculum.

```
265             MD_Init(&m);
266             MD_Update(&m,local_md,MD_DIGEST_LENGTH);
267             k=(st_idx+j)-STATE_SIZE;
268             if (k > 0)
269                     {
270                     MD_Update(&m,&(state[st_idx]),j-k);
271                     MD_Update(&m,&(state[0]),k);
272                     }
273             else
274                     MD_Update(&m,&(state[st_idx]),j);
275
276             /* DO NOT REMOVE THE FOLLOWING CALL TO MD_Update()! */
277             MD_Update(&m,buf,j);
278             /* We know that line may cause programs such as
279                purify and valgrind to complain about use of
280                uninitialized data.  The problem is not, it's
281                with the caller.  Removing that line will make
282                sure you get really bad randomness and thereby
283                other problems such as very insecure keys. */
284
285             MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));
```

Figure 6 More Recent Version of the OpenSSL Source Code

3. *Overly Clever Coding*. Programmers should realize that while very clever coding might save some space and coding time, it is very difficult to understand and is a fertile breeding ground of errors. Figure 6 [44] shows there there are now more calls to MD_Update so the functionality has been separated to some extent, but no information is given about what each call is doing.
4. *Uncritical Use of Automated Software Analysis Tools.* Figure 7 which comes from [18] shows the comparison of code before and after the modification. The right half of the figure has the changes in green. The presentation is a bit deceptive because the effect of the introduced lines is to remove the lines dealing with MD_Update from the program. We need to make it more obvious to an observer that the changes have affected the lines containing MD_Update.
5. *Overwhelming Error Messages.* Thought needs to be given on how to demonstrate all weaknesses in some code without overwhelming the person using the automated tool.
6. *Repairs by Nonexperts.* If the Federal government wants to help improve US cybersecurity it should consider offering code reviews for critical software. Clearly, it is not reasonable for the Federal government to review all code, but Debian and its derivatives such as Ubuntu are very popular and are the basis of a significant number of servers on the Internet. Given the recent Snowden revelations and the deliberate introduction of weaknesses into the BSAFE package by the NSA, it is not clear that one can depend on the Federal government to give a honest appraisal.

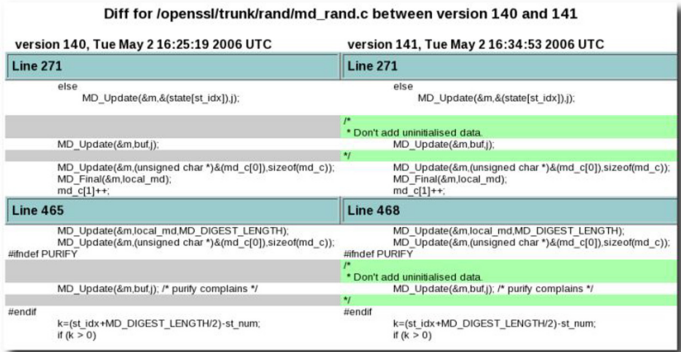

Figure 7 Diff Obscuring High Level Understanding

7. *Ambiguous Communication.* Perhaps a more formal process can be instituted here to make sure that all questions are posed and answered unambiguously. It appears that the standard method of just mailing to a site and having people selectively reply to sections of the e-mail is fraught with danger.
8. *Poorly Distributed and Overly Technical Announcements.* We need to get more people with public relations and communications skills active in the open source community. These people must be made to feel welcome and not put down by the technical community since they have an important job to perform.
9. *Posting Patches Prematurely.* Clearly, it is recommended that organizations not publicly post patches before they announce vulnerabilities. Perhaps they can post patches only to vetted customers to give them a chance to update their systems.
10. *User Community Not Taking Cybersecurity Seriously Enough or Perhaps not Having the Resources to Deal With Critical Issues.* This is a challenging problem that deserves a separate discussion.

We wish to conclude with one additional recommendation. While systems such as Knuth's CWEBB and Beck's Test-Driven Development introduce more formal procedures into code design and creation, there is still room for global formal methods. Books by Holtzmann [52] and Berg, Boebert, Franta and Moher [53] provide a good place to acquire the basics. A current open-source software tool is called Spin and and programmers are encouraged to become become familiar with it [54, 55].

References

[1] Charles Perrow, *Normal Accidents*, Princeton University Press, 1999.
[2] George Markowsky, "Was the 2006 Debian SSL Debacle a System Accient?,"
[3] Donal Knuth, *The Art of Computer Programming*, 3rd ed., 1998, Vol. 2, *Seminumerical Algorithms*.
[4] Geiger counter-based random number generator. http://www.fourmilab.ch/hotbits/.
[5] An atmospherically based random number generator. http://www.random.org/.
[6] Quantum Mechanics Random Number Generator Project, 'http://qrng.physik.hu- berlin.de/.

[7] PQRNG 150 (Quantum Mechanics Random Number Generator) PicoQuant GmBH, http://www.picoquant.com/products/category/ quantum-random-number-generator/pqrng-150-quantum-random-number-generator.

[8] Rodrigo Gallego, Lluis Masanes, Gonzalo De La Torre, Chirag Dhara, Leandro Aolita and Antonio Acfin, "Full Randomness from Arbitrarily Deterministic Events," *Nature Communications* 4, Article number: 2654, http://conference.iiis.tsinghua.edu.cn/QIP2013/wp-content/uploads/2012/12/qip2013_submission_7.pdf

[9] Robert Coveyou, "Random Number Generation Is Too Important to Be Left to Chance," *Studies in Applied Mathematics*, III, 1970, pp. 70–111.

[10] George Markowsky, "Bounds on the Index and Period of a Binary Relation on a Finite Set,"*Semigroup Forum*, v. 13, 1877, pp. 253–259.

[11] "Mersenne Twister," *Wikipedia*, February 1, 2014, http://en.wikipedia.org/wiki/Mersenne_twister.

[12] Makoto Matsumoto and Takuji Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," *ACM Trans. on Modeling and Computer Simulation*, v. 8, n, 1, January, 1998, pp. 3–30.

[13] Jerry Dwyer, 'Quick and Portable Random Number Generators," *Dr. Dobb's Journal*, June 1, 1995, http://www.drdobbs.com/quick-and-portable-random-number- generat/184403024

[14] "RANDU," *Wikipedia*, February 2, 2014, http://en.wikipedia.org/wiki/RANDU

[15] David Ahmad, "Two Years of Broken Crypto," *IEEE Security & Privacy*, 2008, pp. 70–73.

[16] Russ Cox, "Lessons from the Debian/OpenSSL Fiasco,", blog, http://research. swtch.com/openssl.

[17] Bruce Schneier, "Random Number Bug in Debian Linux," *blog Schneier on Security*, http://www.schneier.com/blog/archives/2008/05/random_number_b.html

[18] Luciano Bello and Maximiliano Bertacchini, "Predicatable PRNG In the Vulnerable Debian OpenSSL Package: The What and the How," DEFCON 16, August 8–10, 2008, Las Vegas, NV, http://www.youtube.com/watch?v=yXr7KBC3G3I. The slides are available at http://www.citedef.gob.ar/si6/descargas/openssl-debian-defcon16.pdf.

[19] Jacob Applebaum, Dino Dai Zovi, and Karsten Nohl, "Crippling Crypto: The Debian OpenSSL Debacle,", http://www.youtube.com/watch?v=QdknzkoN_aI. The slides are available at http://trailofbits.files.wordpress.com/2008/07/hope-08- openssl.pdf.

[20] Valgrind Website, http://valgrind.org/.

[21] "Some SecureRandom Thoughts," *Android Developers* Blog, August 14, 2013, http://android-developers.blogspot.com/2013/08/some-securerandom- thoughts.html.

[22] Dan Goodin, "Google Confirms Critical Android Crypto Flaw Used in $5,700 Bitcoin heist,", em Ars Technia, August 14, 2013, http://arstechnica.com/security/ 2013/08/google-confirms-critical-android-crypto-flaw-used-in-5700- bitcoin-heist/.

[23] Jim Edwards, "A Thief Is Attempting To Hide $100 Million In Stolen Bitcoins And You Can Watch It Live Right Now," *Business Insider*, December 3, 2013, http://www.businessinsider.com/a-thief-is-attempting-to-hide-100- million-in-stolen-bitcoins-and-you-can-watch-it-live-right-now-2013–12.

[24] "RSA BSAFE," *Wikipedia*, February 2, 2014, http://en.wikipedia.org/wiki/ RSA_BSAFE.

[25] Matthew Green, "RSA Warns Developers Not to Use RSA Products," blog, http://blog.cryptographyengineering.com/2013/09/rsa-warns- developers-against-its-own.html.

[26] Dan Shumow and Niels Feguson, "On the Possibility of a Back Door in the NIST SP800-90 Dual Ec Prng," http://rump2007.cr.yp.to/15-shumow.pdf.

[27] Glenn Greenwald, "Edward Snowden: the whistleblower behind the NSA surveillance revelations," *The Guardian*, June 9, 2013, http://www.theguardian.com/world/2013/jun/09/edward-snowden-nsa-whistleblower-surveillance.

[28] "Edwward Snowden," *Wikipedia*, February 2, 2014, http://en.wikipedia.org/wiki/Edward_Snowden#cite_note-cnn-hotel-104.

[29] Joseph Menn, "Exclusive: Secret Contract Tied NSA and Security Industry Pioneer," *Reuters*, December 20, 2013, http://www.reuters.com/article/2013/12/20/ us-usa-security-rsa-idUSBRE9BJ1C220131220.

[30] Dan Goodin, "Stop using *NSA-influenced* code in our products, RSA tells customers," *Ars Technica*, September 19, 2013, http://arstechnica.com/security/2013/09/ stop-using-nsa-influence-code-in-our-product-rsa-tells-customers/

[31] Dan Goodin, "More Researchers Join RSA Conference Boycott to Protest $10 million NSA Deal," *Ars Technia*, January 7, 2014, http://arstechnica.com/security/2014/01/more-researchers-join-rsa-conference-boycott-to-protest-10-million-nsa-deal/

[32] Charles Perrow, "Software Failures, Security, and Cyberattacks," Online article at http://www.cl.cam.ac.uk/∼rja14/shb08/perrow.pdf, http://www.tatup-journal.de/english/tatup113_perr11a.php.

[33] "Booby-Trapped Onion Patch Kills Owner, Son,", Los Angeles Times, June 12, 1994, http://articles.latimes.com/1994–06-12/news/mn-3270_1_onion-patch.

[34] "NASA's metric confusion caused Mars orbiter loss," CNN, September 30, 1999, http://www.cnn.com/TECH/space/ 9909/30/mars.metric/.

[35] "Metric Moon,", NASA Science News, online article, http://science1.nasa.gov/science-news/science-at-nasa/2007/08jan_metricmoon/.

[36] Richard Kettlewell, "valgrind-clean the RNG," Debian Bug report logs -#36516, http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=363516.

[37] Kurt Roeckx, "Random number generator, uninitialized data and valgrind,", OpenSSL Mail Archive, http://www.mail-archive.com/openssl-dev@openssl. org/msg21156.html.

[38] Ulf Möller, "Random number generator, uninitialized data and valgrind,", OpenSSL Mail Archive, http://www.mail-archive.com/openssl-dev@openssl.org/ msg21157.html.

[39] Debian Security Advisory 1571, http://www.debian.org/security/2008/dsa-1571.

[40] Luciano Bello, "Exploiting DSA-1571: How to break PFS in SSL with EDH," http://www.lucianobello.com.ar/exploiting_DSA-1571/

[41] Ben Feinstein, "Loaded Dice: SSH Key Exchange & the Debian OpenSSL PRNG Vul- nerability," ToorCon X, September 27, 2008. Slides are available at http://www.cs.unh.edu/it666/reading_list/Keys/ssh_key_exchange.pdf.

[42] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage, "When Private Keys are Public: Results for the 2008 Debian OpenSSL Vulnerability", *ICM'09*, November 4–6, 2009, Chicago Illinois, USA.

[43] "The SSL Keys and Various Applications," http://wiki.debian.org/SSLkeys

[44] OpenSSL Source Code for md rand.c, https://github.com/luvit/openssl/blob/master/openssl/crypto/rand/md_rand.c

[45] TJ. O'Connor, *Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers*, Syngress, Waltham, MA, 2013.
[46] Daniel J. Barrett, Richard E. Silverman, and Robert G. Byrnes, *SSH, The Secure Shell, The Definitive Guide*, 2nd ed., O'Reilly, Sebastopal, CA, 2005.
[47] "A Detector for Known Weak Key Material,", *Debian.org*, http://security.debian.org/project/extra/dowkd/dowkd.pl.gz.
[48] Donald E. Knuth, *Literate Programming*, Center for the Study of Language and Information, 1992.
[49] Donald E. Knuth and Silvio Levy, *The CWEB System of Structured Documentation*, Version 3.6, Addison-Wesley, 1994, http://sunburn.stanford.edu/knuth/cweb. html.
[50] Kent Beck, *Test-Driven Development by Example*, Addison-Wesley, 2002.
[51] "Test-Driven Development," *Wikipedia*, February 4, 2014, http://en.wikipedia.org/wiki/Test-driven_development.
[52] Gerard j. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.
[53] H. K. Berg, W. E. Boebert, W. R. Franta and T. G. Moher, *Formal Methods of Program Verification and Specification*, Prentice Hall 1982.
[54] "Home Page for Spin," http://spinroot.com/spin/whatispin.html.
[55] Gerard J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison- Wesley, 2003.

Biography

George Markowsky spent ten years at the IBM Thomas J. Watson Research Center where he served as Research Staff Member, Technical Assistant to the Director of the Computer Science Department, and Manager of Special Projects. He came to the University of Maine as the first Chair of the Computer

Science Department. During 2004–2005 he was Dean of the American-Ukrainian Faculty at Ternopil National Economic University in Ukraine. In 2006–2007 he was Visiting Professor at Rensselaer Polytechnic Institute Lally School of Management and Technology. He became Chair of the Computer Science Department again in 2008 and served in that capacity until the Department became part of the new School of Computing and Information Science at which time he became the Associate Director of the School. In 2013–2014 he has been a Visiting Scholar in the Department of Computing Security at the Rochester Institute of Technology. He is currently Professor of Computer Science at the University of Maine. George Markowsky has published 113 journal papers, book chapter, book reviews and conference papers on various aspects of Computer Science and Mathematics. He has written or edited 15 books and reports on various aspects of computing. He also holds a patent in the area of Universal Hashing. His interests range from pure mathematics to the application of mathematics and computer science to biological problems. He has also built voice controlled and enhanced keyboard terminals for use by paralyzed individuals. He is very active in homeland security and is the director of the University of Maine Homeland Security Lab.