

Dr. Dobb's Journal

July 2012

Preventing Stack Overflow Attacks



ALSO INSIDE

[Phish Isn't Spam >>](#)

[On Avoiding Being a Bad Programmer >>](#)

[From the Vault:
The Theory, Design, and Implementation of
Cryptographic Hash Functions >>](#)

Dr. Dobb's Journal

CONTENTS

July 2012



COVER STORY

7 Preventing Stack Overflow Attacks

By Oliver Mueller

The use of special prologues and epilogues around functions can easily detect attacks that rely on stack overflows. Here is how “stack smashing” is actually performed and how its occurrence is detected.

17 From the Vault:

The Theory, Design, and Implementation of Cryptographic Hash Functions

By Jesse Walker, Michael E Kounavis, Shay Gueron, and Gary Graunke

This article provides a detailed primer on why hash functions are cryptography’s most widely used primitives.

5 Guest Editorial

By Norman M. Sadeh

In this guest editorial, a Carnegie Mellon computer science professor explains why phish should not be treated as spam.

3 Letters

By you

Readers comment on coding in an echo chamber and bad programmers.

31 Links

Snapshots of the most interesting items on drdobbs.com including the design of LLVM and automatic reference counting on iOS.

32 Editorial and Business Contacts

More on DrDobbs.com

What Makes Great Programmers Different?

The keys to being a good programmer are well known. Greatness, however, requires something else altogether.

<http://www.drdobbs.com/architecture-and-design/240001472>

The Mystery and Magic of Prefix Scan

Prefix Scan might be considered an elemental parallel computation. This operation computes all partial sums of a vector of values resulting in a vector, the same size as the original vector, where each element is the sum of the preceding elements in the original vector up to the corresponding position.

<http://www.drdobbs.com/blogs/parallel/240001744>

C’s New Ease of Use and How the Language Compares with C++

C11 adds useful, handy features that match some of the additions to C++. However, key divergences mean that a few C11 features have no counterparts in C++.

<http://www.drdobbs.com/cpp/240001401>

Disciplined Agile Change Management

Change management can be handled in multiple ways — no one approach is perfect. The best solution depends on several factors, but principally the skill of the project owner and the maturity of the development team.

<http://www.drdobbs.com/architecture-and-design/240001474>

Coding From Within the Echo Chamber

By the content customization of websites and through conscious choices, it’s exceedingly difficult to escape our own echo chamber. Our coding and design suffer for it.

<http://www.drdobbs.com/architecture-and-design/240001127>

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

Mailbag

Readers weigh in on bad programmers and working in an echo chamber

On Avoiding Being a Bad Programmer

In response to Andrew Binstock's editorial "What Makes Bad Programmers Different?" (<http://www.drdobbs.com/architecture-and-design/240001941>), we received many affirmative emails and an interesting query:

I think your brief article was wonderfully accurate. I would only add the following:

- Bad programmers fail to do proper Q/A (that is, they never consider single-stepping the code nor do they perform regression testing).
- Code has a lot of side effects (for example, leaving a flag in one module to affect the operation of another).

- They don't fully understand the full set of use cases for their code (this might be your second bullet on superficial knowledge of domain).

— David Marcus

I am in school to become a computer programmer. My question is this: What do I need to do to be a great programmer? Is there something I need to learn? I am 40-something and starting over. I have self-taught myself some things, just the basic things so far in C++ and Python. Any advice would be most helpful.

[name withheld by editors]

Andrew Binstock responds: Congratulations on your choice of career paths. In theory, you just embarked on many years of enjoyable

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

work! Your question is asked a lot on programmer hangouts — see Reddit (<http://www.reddit.com/r/programming>) and Hacker News (<http://news.ycombinator.com/>). The general advice, with which I completely concur is: 1) Know your tools and know your language well; 2) Write a lot of code; 3) Read a lot of code from developers who are known to be good; 4) Have programmers who are very good look over your code and give you feedback.

On point #2, write lots of programs. They can be short. Start with small utilities, such as a program to dump binary files in to hex on the screen; then add paging to it and a side display of the characters in ASCII form. Another fine utility is to crawl through a directory looking for duplicate files (use a hash code for the file contents and make sure to crawl through all the subdirectories). Progress to building a database of all your friends on Facebook or Twitter. Find out whom they follow and then try to identify which friends you barely know are the closest to your way of thinking. And so on.

As you do this, use tools like lint and valgrind to show you errors or infelicities of your code. And as you work on these projects, look for good code to read. With open source being so accessible today, you'll find lots of good code to review. Truly try to understand what the author is doing. Ask yourself how you'd have solved the problem. Adopt techniques that you like.

Do these things and keep your curiosity, and there's no doubt you will be a very, very good programmer. Being self-taught is no obstacle at all. Once you're confident with your skills and need resume experi-

ence, find a way to contribute to a high-profile open-source project. And you'll be on your way.

Best of luck!

On Avoiding Insularity

With regard to the editorial “Coding from Within The Echo Chamber” (<http://www.drdobbs.com/architecture-and-design/240001127>), we received this thoughtful reminiscence:

My father was an IBM mainframe programmer for many years. I remember the frustration of talking to him about software after coming home from college one year. He would use terminology unlike anything I had ever heard, and after telling him that, he would ask me what the hell are they teaching you in that place?! When I tried to describe to him some of the things that I had learned, he would profess complete ignorance, and then complain that I'm becoming like “one of those Unix weenies.”

[name withheld]

Have a correction or a thoughtful opinion on *Dr. Dobb's* content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of *Dr. Dobb's*.

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

Why Phish Should Not Be Treated As Spam

Phishing — or the use of highly targeted e-mails to induce users to divulge passwords or use malware — is a problem for all companies. In this invited editorial, security expert Norman Sadeh discusses how phishing needs to be treated separately from spam and spam solutions.

By Norman M. Sadeh, Ph.D.

More than 500 million phishing e-mails appear in user inboxes every day. While this number pales in comparison to spam (which accounts for almost 70% of all e-mail traffic), spam is mainly a nuisance, whereas phishing can lead to costly security breaches. In the U.S. alone, phishing attacks on customers have been reported to result in direct financial losses of several billion dollars per year. But for corporations and government organizations, this is just the tip of the iceberg, as more-targeted “spear phishing” attacks can lead to potentially devastating security breaches, loss of sensitive data, and significant financial losses.

Anti-Spam Vendors: Comparing Apples with Oranges

While most anti-spam/anti-virus vendors have repurposed their filters to also catch phishing e-mails, their solutions rely primarily on manually maintained “black lists.” To minimize the risk of flagging legitimate sites, these black lists typically come in the form of fraudulent URLs that are manually vetted by people. By their very nature, these black lists are al-

ways one step behind, lagging by at least several critical hours and sometimes days. During that lag time, many phishing e-mails will go undetected by spam filters and many of the malicious websites to which phishing victims are directed will not be flagged by their browsers — as the browsers rely heavily on black lists, too. Yet studies have shown that during regular work hours, 50% of users who fall for phishing attacks read their e-mail within two hours of the time it reaches their inbox. This number reaches 90% within eight hours. In other words, a lag in updating black lists by just a few hours can have devastating consequences. “Reply-to” phishing e-mails with no attachments and no links are another example of phishing attacks that often go undetected by anti-spam/anti-virus filters. This is due in part to anti-spam filters’ use of simple “bag of words” techniques. These are techniques that look for e-mails containing collections of words that are indicative of spam. They are good for catching spam, but they are unable to differentiate between phishing e-mails and legitimate e-mails, since many phishing e-mails are crafted to look just like legitimate e-mails.

Ironically, this state of affairs is not entirely obvious to someone who looks at the statistics advertised by many vendors promoting their anti-

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

virus/anti-spam filters. Many continue to boast about their ability to catch “up to 99%” of malicious e-mail, a confusing statement that clumps together spam, viruses, and phishing attacks. Because almost 70% of all e-mail traffic is spam, and phishing attacks amount to only about 0.5% of the traffic, “catching up to 99% of malicious e-mail” is an ambiguous statement. The consequences of finding an unfiltered spam e-mail in your inbox cannot be equated to the potential consequences of receiving a phishing e-mail. In other words, spam vendors are often comparing apples with oranges. In addition, they fail to tell us about the number of false positives they may end up flagging to reach the 99% performance they boast of. False positives are those legitimate e-mails they sometimes classify as spam and move to your junk box, forcing you to regularly go and check whether an important e-mail found its way there by mistake. In truth, to reach 99% effectiveness, many spam filters require settings that also lead to more false positives, effectively reducing the value of the filter because users are forced to regularly check the content of their junk box for legitimate correspondence.

Phishing Attacks That Make It Past Existing Filters Are the Ones Users Most Likely Fall For

Even when it comes to phishing, not all e-mails are created equal. Studies have shown that high-volume phishing campaigns claiming to come from well-established organizations such as large banks, ISPs, or the IRS, are those that people are least likely to fall for. In contrast, targeted phishing e-mails, or “spear phishing” e-mails, which are directed at small groups of people such as employees of a particular department or even individuals, tend to be much more effective at fooling their recipients. These e-mails have been used to initiate many of the high profile security breaches reported over the past couple of years, as well as many

lower profile attacks on smaller organizations. Statistics that simply look at percentages of phishing e-mails caught (including many of the easy-to-detect, high-volume phishing e-mails) fail to recognize these complexities and therefore produce seemingly reassuring numbers that are skewed towards the least dangerous types of phishing e-mails.

A Call for Better Benchmarks and What To Do While We Wait

Given this state of affairs, it is high time for industry and independent evaluation organizations to come up with benchmarks that reflect the significantly higher risks associated with phishing attacks. Simply focusing on spam and viruses is no longer a fair assessment of potential risks. We need benchmarks that look separately at phishing and tests that reflect the lag found in many filtering solutions. Testing a filter’s response to a phishing attack days or even weeks after that phishing attack was first launched is not going to properly inform prospective customers about the true performance of the filter. In the meantime, organizations evaluating e-mail filtering solutions should make sure to request trial licenses and evaluate these solutions on live e-mail while looking specifically at the anti-phishing performance of the filters over periods of at least several weeks. Those who cannot afford to conduct such tests should specifically request information on the effectiveness of the solution when it comes to catching phishing e-mail.

— *Dr. Norman Sadeh is a Professor of Computer Science at Carnegie Mellon University. He is also cofounder and chief scientist of Wombat Security Technologies, a company that has commercialized a suite of cyber security training software products and anti-phishing filtering products.*

[Comment](#)

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

Preventing Stack Overflow Attacks

The use of special prologues and epilogues around functions can easily detect attacks that rely on stack overflows. Here is how “stack smashing” is actually performed and how its occurrence is detected.

By Oliver Mueller

Even today, buffer overflows are a threat to computer-system security. Over the last few years, vendors of operating systems have set up their products with different strategies to prevent buffer overflow attacks. Address space layout randomization (ASLR) and non-executable stacks are two such improvements that try to prevent attackers from using vulnerabilities based on buffer overflows for a successful intrusion.

In addition, compiler vendors added techniques against buffer overflows, especially stack overflows, to their products. All major vendors of C and C++ compilers have added detectors to their compilers that can sense stack smashing — the willful use of stack overflows to gain control of a system — before any harm can be done. They cannot prevent the stack from being smashed, but they can detect the smashing and avoid executing the malware payload. Such “buffer protectors” include Stack Smashing Protector (SSP) for GNU’s gcc, ProPolice for IBM’s XLC, and Buffer Security Check for Microsoft’s Visual compilers (option /GS). In this article, I examine in detail how stack overflows occur, how

stack smashing attacks work, and how they can be detected. I also examine the implementation of the GNU SSP and show the methods it employs to thwart these attacks.

Understanding the Problem

Before discussing a solution, you have to understand the problem: What happens in the case of a stack overflow and why is it so dangerous?

Whenever a program calls a subroutine by a CPU instruction `CALL` or `BRANCH`, it saves the current instruction pointer (IP) onto the stack. It marks its position before it branches, so it knows where to return after termination of the subroutine. This saved address on the stack is called a return address.

A high-level programming language like C or C++ also puts local variables of the subroutine on top of the stack (Figure 1). Thus, the subroutine gets its own memory area on the stack where it can store its own data. This principle is also the key of recursive routine calls

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

because every new call to the subroutine gets its own return address and its own local variables on the stack.

Upon termination of the subroutine, high-level languages first clean up the stack by removing all local variables. After that, the stack pointer (SP) again points to the saved return address. At a RET or RETURN instruction, the processor reads the return address from stack, jumps back to the former IP position, and continues the original program flow.

This flexible layout allows theoretically infinite levels of recursion, but several problems are also inherent. If a program does not check the boundaries of local variables at writing, it could write beyond the intended space of these variables. In the best case (from a operating system's perspective), this will overwrite other variables. In the worst case, it will overwrite the return address. Why? On most systems, including state-of-the-art systems, the stack and heap of a program are growing toward the same point in memory. Where the heap grows from lower to higher addresses, the stack grows from higher to lower addresses (see Figure 2). In other words: The stack stands upside down.

Even if the stack grows from higher to lower addresses, the local variables on the stack work the "normal" way: They grow from lower to higher addresses. Any string and any array store their values of lower indexes at lower addresses and higher ones at higher addresses.

Let's assume a program writes data to a local string variable, by `strcpy()` for example. `strcpy()` does not check for any boundary. It copies data to the destination variable until it reads a null byte from the source variable. Thus, it is able to write more bytes onto the stack than there is space dedicated to the provided string variable. As a result, it might overwrite other data in higher addresses in memory and therefore below the destination variable on "lower ad-

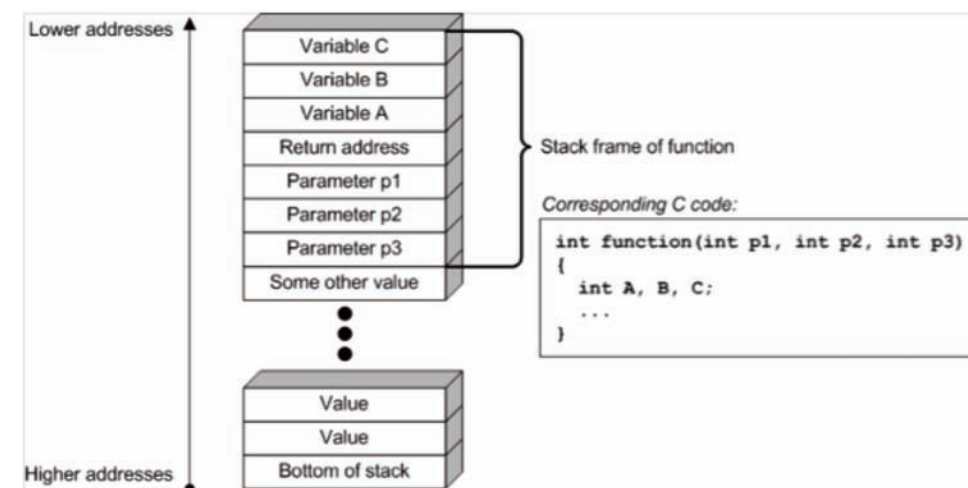


Figure 1: Stack layout.

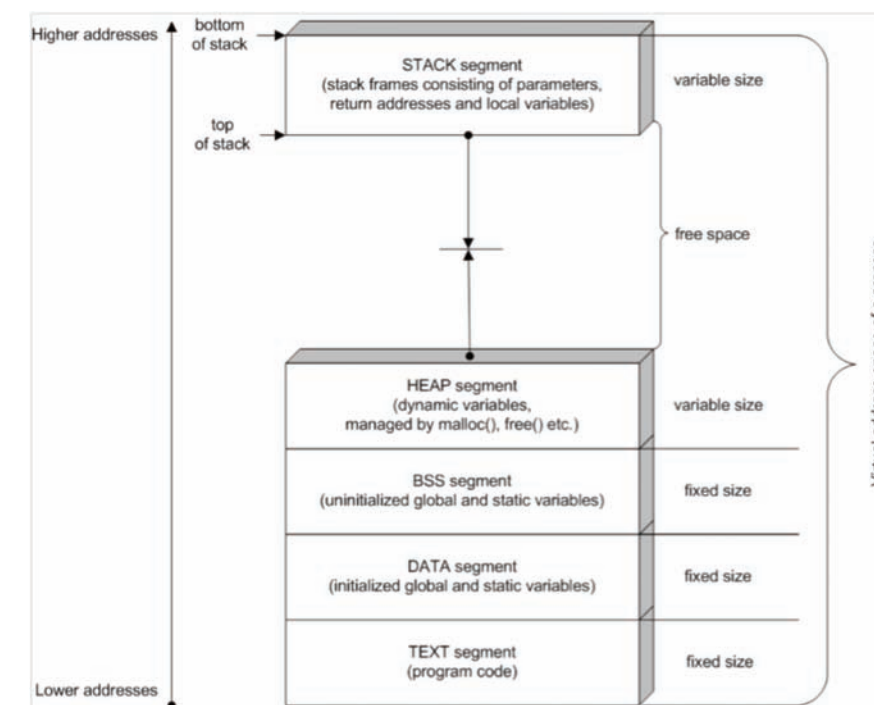


Figure 2: The memory layout of a process.

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

addresses” on the stack (see Figure 3). If the copy process is not stopped, sooner or later, the program will overwrite the return address, too. The return address will be destroyed, or more precisely, given a new value.

At termination of the subroutine, when the program retrieves the return address, that address has changed. Under normal circumstances, the program flow jumps to an unpredictable address. Chances are very good that the jump leads into a segment outside the program’s own memory. The system detects a segmentation violation and kills the process with signal number 11 (SIGSEGV). The program crashes and leaves a core file.

So far, so good. But what happens if the target of the changed return address is not outside the program’s own memory? In this case, the jump might succeed and execution will go on. If there is valid machine code, it will be executed. If not, the processor will create an illegal instruction exception.

Footprint of a Stack Overflow Attack

Attackers change the return address via a buffer overflow — not to an unpredictable location, but to a specific address. In other words: They can control where the jump goes. A classic attack includes a so-called “payload” (also called a “cuckoo’s egg”) in the overflowing data, which consists of three parts:

1. An NOP sled
2. A shell code
3. A new return address

The NOP sled consists of a repeated processor instruction, called NOP

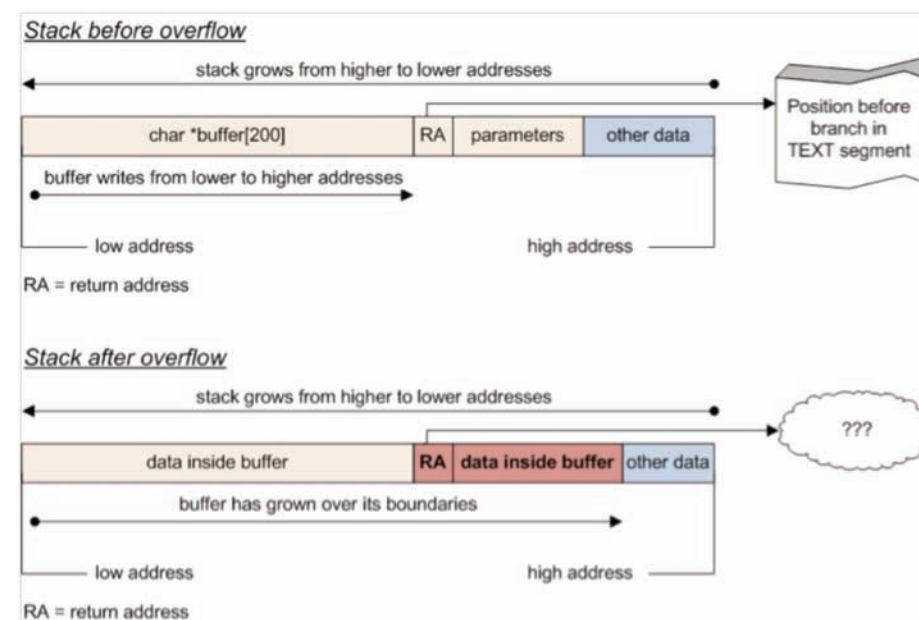


Figure 3: A stack overflow.

(no operation). This instruction does nothing but increase the IP by one. In other words, it’s a do-nothing filler byte.

The shell code is a little program written by the attacker in assembly language and machine code, respectively. Normally, it utilizes system calls to gain a root shell. This is why it is called “shell” code.

Last but not least, the new return address is provided, which points somewhere into the NOP sled. If the subroutine terminates, the program flow will jump into the NOP sled and execute the NOPs until it reaches the shell code (see Figure 4). The shell code opens a root shell and the attacker gains control over the system.

The NOP sled is simply a way to avoid knowing the precise address of the shell code on the stack. Systems differ, and calculating the exact return address is an art of its own. It is easier to jump somewhere into the middle of the NOP sled. Stack layout will differ from system to sys-

IN THIS ISSUE

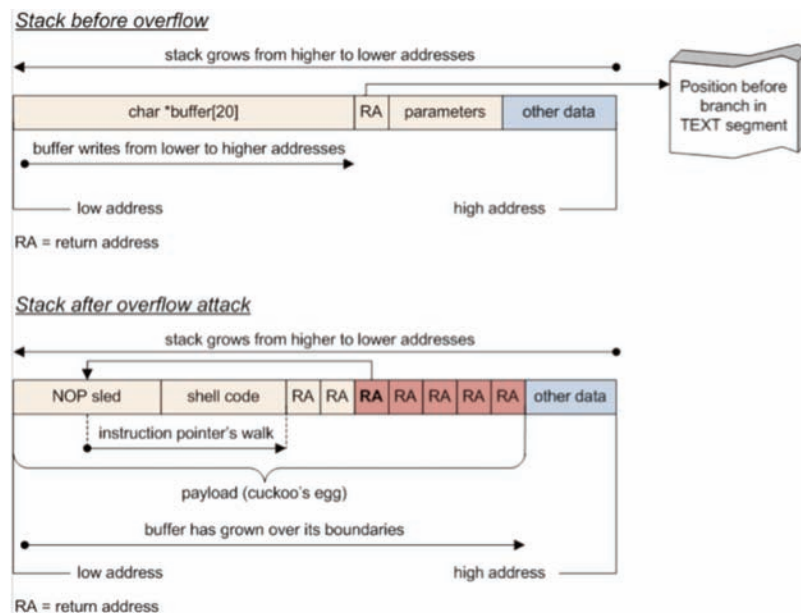
[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

Figure 4: A stack overflow attack.

tem and from program execution to execution. It won't matter if the payload lays a bit higher or lower on the stack. As long as the jump goes inside the NOP sled, the exploit will work. For the same reason, the return address is not simply one single value. It is a concatenation of repetitions of the intended return address. One of these repeated addresses will overwrite the original return address on the stack.

Attacks with shell code in the stack segment don't work on contemporary systems anymore. Features like non-executable stacks, provided by the hardware or operating system, prevent execution of injected code on the stack. But the threat is not yet removed entirely. There are other tricks that an attacker can use. The "Return to libc" is one such trick. Such attacks also rely on stack overflows, but they are a bit more complicated. For a stack smashing protector, it makes no difference whether it is a classic or modern attack. So I'll concentrate on classic attacks for the sake of simplicity.

Protecting Canaries

The classic stack overflow attack and most of its modern successors are organized in two steps. The first step is a preparation stage in which the payload overwrites the return address. After this, the attacker has to wait while the "normal" algorithm of the vulnerable subroutine goes on. Only when the subroutine terminates will the prepared return address force the CPU to start the real attack with its harmful impact.

Thus, there is a temporal and operational gap between the preparation and the implicit invocation of the injected code. For a protection mechanism, this gap is predestined to get a closer look on the stack before returning from the subroutine. In no case can an attacker change the code of the subroutine itself. So the subroutine's code stays always intact.

The aforementioned stack smashing protectors of contemporary compilers use exactly these two findings — the gap and the untouched code of the subroutine in this gap. They add additional code to the subroutine as a prologue and an epilogue that detect stack overflows.

A typical subroutine of a language like C or C++ consists of four parts at the machine level:

1. Initialization: the preparation of space on the stack for local variables.
2. Subroutine body: the subroutine's implemented algorithm.
3. Clean-up: removing local variables from the stack.
4. Return: jump back to original address before the branch.

Stack smashing protectors (SSP) add two additional parts to this schema. A special prologue, which is placed before the initialization,

IN THIS ISSUE

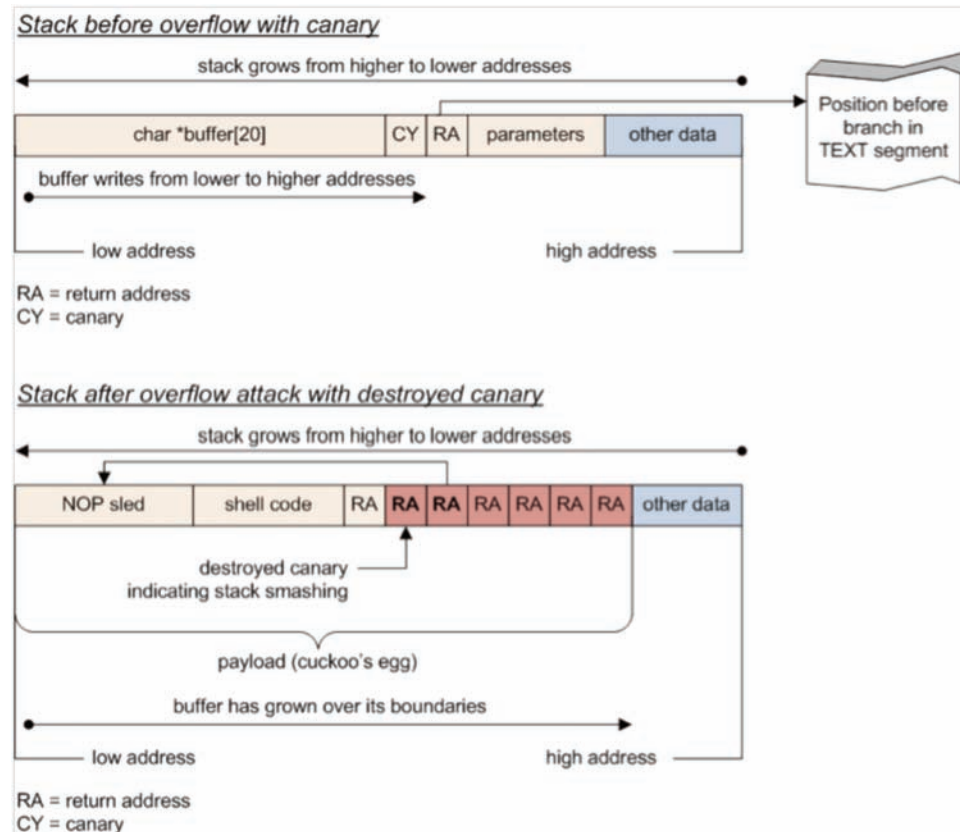
[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

Figure 5: Destroyed canary indicating stack smashing.

and an epilogue, which fits between clean-up and return. This leads to the following new layout of a subroutine:

1. SSP's prolog
2. Initialization
3. Clean-up
4. SSP's epilog
5. Return

The idea is to add a barrier on the stack between the return address and the local variables. In case of a buffer overflow on the stack, the barrier will be crossed and destroyed. SSP's epilog will detect this invalid barrier. The barrier is simply an integer value and is called a "canary" in SSP's own wording. Before the return address is destroyed, the canary will be destroyed (see Figure 5).

The destroyed canary indicates stack smashing. A simple comparison with the SSP's epilog shows the changed canary and uncovers the overflow and a potential attack, respectively. All this is detected before any branch based upon whether a changed return address might occur. The program doesn't lose control and it can stop its process in the SSP epilog before the return step and before any potential damage is done.

Practical View

gcc's Stack Smashing Protector (SSP) was developed by IBM and originally called ProPolice. It is available as a patch for gcc 3.x and has been included by default in gcc since version 4.1. The command line flag `-fstack-protector` switches SSP on. `-fno-stack-protector` switches it off. If nothing is specified at the invocation of gcc, whether SSP is switched on or off by default depends on the system and the configuration of gcc. Some systems (like OpenBSD) switch SSP on. Others (like Debian GNU/Linux) switch it off.

For a practical view, I will use Debian GNU/Linux 6.0.1 with gcc 4.4.5 on a x86_64 system. The test program is `vuln.c` (Listing One). It suffers from a vulnerability in function `vuln()`. It uses `strcpy()` without any previous boundary checks. The buffer `buf` overflows if more than 20 bytes are provided via parameter `str`.

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)**Listing One**

```
#include <string.h>

void vuln(const char *str)
{
    char buf[20];
    strcpy(buf, str);
}

int main(int argc, char *argv[])
{
    vuln(argv[1]);
    return 0;
}
```

Depending on the grade of destruction, the program may print a back-trace and perhaps a memory dump. But in all cases, the program detects stack smashing and terminates itself.

Compiler-Added Code

Let's look at the generated assembly code with and without enabled SSP. The following commands generate assembly language code:

```
gcc -S -o vuln-without-ssp.s -fno-stack-protector vuln.c
gcc -S -o vuln-with-ssp.s -fstack-protector vuln.c
```

Listings Two and Three show commented versions of the generated assembly code for function `vuln()`.

Listing Two

```
vuln:
.LFB0:
    .cfi_startproc
    pushq   %rbp                ; current base pointer onto stack
    .cfi_def_cfa_offset 16
```

```
    movq    %rsp, %rbp          ; stack pointer becomes new base pointer
    .cfi_offset 6, -16
    .cfi_def_cfa_register 6
    subq    $48, %rsp           ; reserve space for
                                ; local variables on stack

                                ; bring arguments from registers onto stack
    movq    %rdi, -40(%rbp)     ; 1st argument from rdi to stack

                                ; prepare parameters for strcpy()
    movq    -40(%rbp), %rdx     ; 1st argument to rdx
    leaq    -32(%rbp), %rax     ; 2nd argument to rax

                                ; call strcpy()
    movq    %rdx, %rsi          ; source address from rdx to rsi
    movq    %rax, %rdi          ; destination address from rax to rdi
    call    strcpy              ; call strcpy()

    leave                       ; clean-up stack
    ret                                           ; return
    .cfi_endproc
```

Listing Two shows no surprises. After mandatory preparations, like saving the current content of the base pointer register, the program prepares the stack for local variables. Linux's typical juggling of arguments given in registers makes the code look a bit more complex. Before calling `strcpy()`, the source and destination addresses go into `rsi` and `rdi` where Linux' `strcpy()` expects them. When `strcpy()` terminates, the program cleans up the stack and jumps back to the return address. Listing Two is a typical assembly listing for code generated by a C compiler on Linux.

Listing Three

```
vuln:
.LFB0:
    .cfi_startproc
    pushq   %rbp                ; current base pointer onto stack
```

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

```

.cfi_def_cfa_offset 16
movq   %rsp, %rbp      ; stack pointer becomes new base pointer
.cfi_offset 6, -16
.cfi_def_cfa_register 6
subq   $48, %rsp        ; reserve space for
                        ; local variables on stack

                        ; bring arguments from registers onto stack
movq   %rdi, -40(%rbp)  ; 1st argument from rdi to stack

                        ; SSP's prolog: put canary onto stack
movq   %fs:40, %rax      ; canary from %fs:40 to rax
movq   %rax, -8(%rbp)    ; canary from rax onto stack
xorl   %eax, %eax        ; set rax to zero

                        ; prepare parameters for strcpy()
movq   -40(%rbp), %rdx   ; 1st argument to rdx
leaq   -32(%rbp), %rax   ; 2nd argument to rax

                        ; call strcpy()
movq   %rdx, %rsi        ; source address from rdx to rsi
movq   %rax, %rdi        ; destination address from rax to rdi
call   strcpy            ; call strcpy()

                        ; SSP's epilog: check canary
movq   -8(%rbp), %rax    ; canary from stack to rax
xorq   %fs:40, %rax      ; original canary XOR rax
je     .L3               ; if no overflow -> XOR results in zero
                        ;                               => jump to label .L3
                        ; if overflow -> XOR results in non-zero
call   __stack_chk_fail  ;                               => call __stack_chk_fail()

.L3:
leave          ; clean-up stack
ret            ; return
.cfi_endproc

```



dtSearch®

Instantly Search Terabytes Of Text

- 25+ fielded & full-text search options
- dtSearch's own file parsers **highlight hits** in popular file & email types
- Spider supports static & dynamic data
- APIs for .NET, Java, C++, SQL, etc.
- Win / Linux (64-bit & 32-bit)

"Lightning Fast" – *Redmond Mag*

"Covers all data sources" – *eWeek*

"Returns results in less than a second" – *InfoWorld*

www.dtSearch.com
Fully-Functional Evaluations

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

Listing Three is similar to Listing Two. The implemented functions are the same, but Listing Three adds SSP to the code. SSP's prolog starts with the first `movq` statement, adds a canary prototype from address `fs:40` into register `rax`, and from there onto stack. After these steps, register `rax` is set to zero (by XORing it with itself).

After the subroutine's body is the SSP's epilogue. It reads the canary from the stack and stores it in register `rax`. Then it calculates `fs:40 XOR rax`. In other words: It XORs the canary prototype with the canary from the stack. If this operation results in zero, everything is fine. `fs:40` and the value in `rax` are then known to be identical. No stack overflow has occurred that destroyed the canary or the return address. If the result of the XOR operation returns a non-zero result, the canary on the stack has been changed and a buffer overflow has indeed occurred.

This result, non-zero or zero, sets or resets the zero flag in the CPU's status register. From there, it is simple to use a conditional jump. In the case of gcc's SSP, it's a `je` (jump if equal/zero), which does the trick. If the result is zero, the program jumps to label `.L3` and subroutine `__stack_chk_fail` won't be called. The program performs the stack clean-up and jumps back to the return address.

On a non-zero result of the XOR operation, no jump is executed. `__stack_chk_fail` is called. This subroutine handles the detected overflow and terminates the process. This scheme of prolog/epilog and canary is very simple but effective.

SSP in Action

Let's have a look at this situation in a debugger. We start a debugging session on our previously built `vuln-with-ssp`:

```
gdb -q ./vuln-with-ssp
```

On the (gdb) prompt, a command list will show the following:

```
(gdb) list
2
3     void vuln(const char *str)
4     {
5         char buf[20];
6         strcpy(buf, str);
7     }
8
9     int main(int argc, char *argv[])
10    {
11        vuln(argv[1]);
```

With `break 5` we set a break point at line 5. Now I'll start the program with an overflowing argument:

```
run 'perl -e 'print "X"x40''
```

The program starts and stops immediately at line 6 because of the break point. The stack is at that time completely initialized but "un-smashed" because the dangerous `strcpy()` is not yet invoked.

Listing Three tells us in line `movq %rax, -8(%rbp)` that the canary is addressed -8 bytes relative to the base pointer (register `rbp`). So we can list this position in memory by invoking `x/2 $rbp-8` in our gdb session:

```
(gdb) x/2x $rbp-8
0x7fffffff5e8: 0xc4671b00      0x0c3b5a5b
```

The whole stack frame of subroutine `vuln()` has, according to line `subq $48, %rsp` of Listing Three, a size of \$48 bytes (decimal: 72 bytes). So let's list the whole stack frame:

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

```
(gdb) x/18x $rbp-72
0x7fffffff5a8: 0xf7dd5750      0x00007fff      0x00008000
0x00000000

0x7fffffff5b8: 0x004005e0      0x00000000      0x00000000
0x00000000

0x7fffffff5c8: 0xffffe92e      0x00007fff      0x00000000
0x00000000

0x7fffffff5d8: 0x00400625      0x00000000      0xf7a8daa8
0x00007fff

0x7fffffff5e8: 0xc4671b00      0x0c3b5a5b
```

The last line shows the canary again. Also interesting is the current stack trace because it gives us an insight into the return address. The `bt` command let us list it:

```
(gdb) bt
#0  vuln (str=0xffffffffe92e 'X' ) at vuln.c:6
#1  0x0000000004005ba in main (argc=2, argv=0xffffffffe6f8)
    at vuln.c:11
```

Besides the argument (`str=0xffffffffe92e`), which holds the string with 40 times `x`, we see the address in the `main()` function from which `vuln()` was called. The address is `0x0000000004005ba`. A look at the base pointer's address also tells us that value is the return address:

```
(gdb) x/4x $rbp
0x7fffffff5f0: 0xffffe610      0x00007fff      0x004005ba
0x00000000
```

If we take into account that we are on a little-endian system (an x86 processor), the last two values, `0x004005ba` and `0x00000000`, show us the saved return address as well. Now I'll execute the `strcpy()` statement with `gdb`'s command `step`. `strcpy()` copies the given string of 40 times `x` into the much-too-little buffer `buf`. The stack trace now looks different:

```
(gdb) bt
#0  0x00007ffff7af9560 in strcpy () from /lib/libc.so.6
#1  0x00007ffff7dec4ef in ?? () from /lib64/ld-linux-x86-64.so.2
#2  0x00007ffff7df22f5 in ?? () from /lib64/ld-linux-x86-64.so.2
#3  0x0000000000400582 in vuln (str=0xffffffffe92e 'X' )
    at vuln.c:6
#4  0x00000000004005ba in main (argc=2, argv=0xffffffffe6f8)
    at vuln.c:11
(gdb) x/4x $rbp
0x7fffffff5f0: 0x58585858      0x58585858      0x58585858
0x58585858
```

The return address was overwritten, but so was the canary:

```
(gdb) x/2x $rbp-8
0x7fffffff5e8: 0x58585858      0x58585858
```

Command `c` lets me continue the program and SSP detects the smashing:

```
(gdb) c
Continuing.
*** stack smashing detected ***: /home/omuegger/ssp/vuln-with-ssp terminated
===== Backtrace: =====
/lib/libc.so.6(__fortify_fail+0x37) [0x7ffff7b61667]
/lib/libc.so.6(__fortify_fail+0x0) [0x7ffff7b61630]
/home/omuegger/ssp/vuln-with-ssp[0x400596]
/home/omuegger/ssp/vuln-with-ssp[0x400500]
===== Memory map: =====
...
```

Where Does the Canary Come From?

The canary's prototype is read from `%fs:40` as seen in Listing Three. But where does it come from? Where does the initialization originate?

There is a function `__guard_setup()` provided by `gcc` that sets up the canary prototype. The code is similar to this:

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

```
fd = open ("/dev/urandom", 0);
if (fd != -1) {
    ssize_t size = read (fd, &__guard, sizeof(__guard));
    close (fd) ;
    if (size == sizeof(__guard)) return;
}
/* If a random generator can't be used, the protector switches the
guard
to the "terminator canary" */
__guard[0] = 0; __guard[1] = 0; __guard[2] = '\n'; __guard[3] = 255;
```

The canary is read from `/dev/random`, and changes from execution to execution. Thus, it's hard to guess for an attacker (one cannot simply include the canary with a payload).

If `/dev/random` is not available, the SSP fills the prototype with terminator characters, which cannot be transferred by a payload. `strcpy()`, for example, would expect the end of string while reading a null byte.

The use of prologues and epilogues to create a canary to detect intentional stack overflows is one of the key tools in preventing this kind of attack.

— *Oliver Mueller (<http://oliver-mueller.com/>) works as freelance IT consultant, IT trainer and IT manager. His preferred subjects are: software engineering, UNIX, Linux, Java EE, and mobile platforms, as well as integration and migration of legacy systems (z/OS, OpenVMS).*

[Comment](#)

SHRINKWRAP YOUR ANDROID APPS

WITH THE ADDED SECURITY OF BRAND-NAME CERTIFICATION

Symantec, the world leader in authentication services, introduces Symantec™ Code Signing for Android. Now, you can digitally sign and optimize .apk files for the Android platform. Plus, manage certificate keys and applications for easy application version updates in Google Play, upload an app image, and access full reporting of signing activity—all within the Symantec Code Signing Portal.

 Read [Protecting Your Android Applications with Secure Code Signing Certificates](#) or call 1-866-893-6565 to see how Symantec Code Signing can help make your Android applications more trusted and adopted.

 Website Security Solutions

Copyright © 2012 Symantec Corporation. All rights reserved. Symantec, the Symantec Logo, and the Checkmark Logo are trademarks or registered trademarks of Symantec Corporation or its affiliates in the U.S. and other countries. Other names may be trademarks of their respective owners.

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

From the Vault

The Theory, Design, and Implementation of Cryptographic Hash Functions

Back in 2009, we explored why hash functions are cryptography's most widely used primitives.

— DDJ

By **Jesse Walker, Michael E Kounavis, Shay Gueron, and Gary Graunke**

Hash functions are one of cryptography's most fundamental building blocks, even more so than encryption functions. For example, hash functions are used for digital fingerprinting and commitment schemes, such as message authentication and random number generation, as well as for digital signature schemes, stream ciphers, and random oracles.

Recently, Antoine Joux, one of the leading cryptographers of our time, discovered multi-collision attacks against the general framework in which hash functions are constructed [1], and Xiaoyun Wang created an attack that breaks the collision-resistance property of the most widely deployed hash functions [2], including MD5 and SHA-1. In 2005, Arjen Lenstra and Wang demonstrated how to forge two digital certificates, based on the MD5 hash function, by using different keys but the same signature — something that was hitherto thought to be im-

possible [3]. These attacks that were discovered, and other vulnerabilities that were exposed by researchers, have stimulated a resurgence of research into hash functions. This research has also spawned an international competition, sponsored by the National Institute of Standards and Technology (NIST), an agency responsible for standards used by the U.S. Government, to create a next-generation hash function design (<http://www.nist.gov/index.html>).

In this article, we highlight some recent work in hash function development. We begin by providing some background on hash functions and look at the problems that hash functions address. We then sketch how to build a hash function. Moving on, we outline recent seminal work in the field of hash functions. We describe two radically different designs entered in the NIST competition, the Skein and Vortex designs, created in part with Intel participation. This is followed by a discussion

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

of the design rationale for each where we also compare and contrast the basic design decisions. We end with a summary of our findings.

Hash Functions

A hash function is usually defined as a function H satisfying three properties [4]:

- **Collision resistance.** It is computationally infeasible to find two distinct bit strings $s \neq s'$ such that $H(s) = H(s')$.
- **Pre-image resistance.** Given a hash value t in the range of H , it is computationally infeasible to find a string s for which $H(s) = t$.
- **2nd Pre-image resistance.** Given a string s and hash value t such that $H(s) = t$, it is computationally infeasible to find a second string s' such that $H(s') = t$ as well.

A hash function maps the set of all bit strings into a “message digest” of defined length, called the hash function’s “block size.” Since there are many more strings than message digests, at least one digest output by the hash function must be the image of more than one input string. It is therefore remarkable that it is possible to build a function h that has the three properties previously noted. These properties imply that h essentially acts like a randomly selected compression function.

It is instructive to describe some typical use cases:

- **Digital fingerprinting.** Hash functions construct effective digital fingerprints. If d represents a digital data structure, such as a document, then the message digest $h(d)$ is its fingerprint; the 2nd-pre-image-resistance property says it is infeasible to find a second data structure d' with the same fingerprint $h(d') = h(d)$.

- **Digital signatures.** Digital signatures extend digital fingerprinting by encrypting the hash value $h(d)$ of a data structure d under a private key.
- **Message authentication.** A hash function h used with a secret key K can be used to authenticate a message m . The idea is to create a tag $t = h(K || m)$, where “||” denotes string concatenation, which is sent with the message and verified by the receiver. (This does not quite work in practice, because the cascade construction introduces vulnerabilities hashing the last message block. Instead the tag is essentially computed as $h(K || h(K || m))$). Because of pre-image resistance, it is infeasible for an attacker to create the same tag t unless he or she knows the key K , and because of collision resistance, the tag could have been created only by concatenating the message m to the key K .
- **Pseudo-random number generation.** One standard way to build a random number generator is to take a key K , usually called a “seed,” and to compute $h(K || 0)$, $h(K || 1)$, $h(K || 2)$, with each digest representing a different random number. If h and K are carefully selected, it is infeasible to distinguish this, by any statistical test, from a stream of genuine random numbers.
- **Stream ciphers.** A stream cipher can be built by taking a hash function h , and encrypting message m_i by $m_i \oplus h(K, i)$, where “ $\oplus$ ” denotes XOR, and decryption is $m_i \oplus h(K, i) \oplus (m_i \oplus h(K, i)) \oplus h(K, i) = m_i$.
- **Random oracles.** Random oracles can be thought of as specialized random number generators. They are used widely in cryptography, such as for randomizing public and private key encryption, to make them secure from arcane attacks.

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

Designing Hash Functions

The standard approach to building a hash function is first to construct a compression function that operates on the input strings of a fixed length, and then to use the cascade construction to extend the compression function to strings of arbitrary length [5, 6].

Compression functions are usually built out of block ciphers. Recall that a block cipher is a pair of D and E functions, for decrypting and encrypting, respectively, that operate on strings of a particular length, called the “block size.” If the block size is n -bits, then the encryption of an n -bit string s is $E(s)$, and its decryption is $D(s)$. Every string can be encrypted or decrypted, and $E(D(s)) = D(E(s)) = s$, meaning E (and D), is a permutation of the set of all n -bit strings. Cryptographers say a block cipher is secure if both $s \rightarrow E(s)$ and $s \rightarrow D(s)$ are indistinguishable from a randomly selected permutation. To meet the indistinguishability property, block ciphers are keyed, so a block cipher represents a family of permutations. A particular block cipher instance is selected by choosing a key K . The resulting encryption and decryption instances are denoted E_K and D_K . That is, for each choice of a key K , $s \rightarrow E_K(s)$ (and $s \rightarrow D_K(s)$) behave like different randomly selected permutations.

Compression Functions

The compression functions for all the hash functions commonly used today are built in the following way:

- Select a block cipher scheme (E, D) .
- Define a compression function $c(iv, s) = E(s \oplus iv) \oplus iv$.

Here s denotes a message of exactly n -bits, and iv denotes an initialization vector. This recipe for c says to use s as the encryption key and iv as the data to be encrypted, and then to use XOR s with the encrypted result $E(s \oplus iv)$. The mapping $(iv, s) \rightarrow E(s \oplus iv) \oplus iv$ is called the Davies-Meyer construction [7] for E . It is easy to show that a block cipher used in Davies-Meyer mode is collision-resistant, pre-image resistant, and 2nd pre-image resistant. c is called a compression function because it compresses $\langle iv, s \rangle$ into a new string iv' of exactly s 's length. Other compression function constructions also exist: both Vortex and Skein use the Matyas-Meyer-Oseas [8] construction, $c(iv, s) \rightarrow E(iv \oplus s) \oplus s$, which is identical to Davies-Meyer, except it reverses the role of iv and s .

The Cascade Construction

The cascade construction builds a hash function h from a compression function c with block size n as follows:

```

cascade (s)
  pad (s);  $s_1 s_2 \dots s_b \leftarrow s$ ;  $iv_1 \leftarrow iv$ ; do  $i = 1$  to  $b$  do  $iv_{i+1} \leftarrow c(iv_i, s_i)$  od;
  output  $iv_{b+1}$ 

```

Every hash function based on a block cipher must define a padding scheme, because compression functions only operate on strings s of length n bits exactly. Most padding schemes pad s with a single 1 bit followed by as many 0 bits as are necessary to bring the length to a multiple of n . The length of the unpadded string s is then encoded as an n -bit integer and appended to defend against extension attacks.

Once padded, partition s into $b = \lceil |s|/n \rceil$ blocks, each consisting of n bits ($|s|$ denotes s 's length in bits): $s_1 s_2 \dots s_b \leftarrow s$.

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

Finally, beginning with a hash-function-specific initialization vector iv , serially compute $c(iv_i, s_i)$ for each block s_i .

The cascade construction extends the collision resistance, pre-image resistance, and pre-image resistance properties from a compression function to a function operating on strings of arbitrary length [9]. The cascade construction is sometimes called the Merkle-Damgard construction after its inventors. This construction is intrinsically serial, as it needs to be able to detect problems, such as two blocks being exchanged.

Hashing Today

We just summarized the state of the art during the early part of this decade, prior to two significant publications. The first was by Antoine Joux, who introduced the multi-collision attack. The second was by Xiaoyun Wang, where she described an attack, based on differential cryptanalysis, against all of the hash algorithms broadly used today.

Suppose a hash function is built out of a compression function by using the cascade construction. Also suppose that someone has broken the collision resistance of the hash function; that is, they have discovered two distinct strings $s \neq s'$ so that $h(s) = h(s')$. Joux observed that it is easy to find many more collisions for little additional cost [1]. The source of the problem is that the cascade construction maintains too little state as it progresses from one invocation of the compression function to the next. Joux's result says that by itself the cascade construction is too weak to serve as an adequate building block for constructing hash functions.

Wang's attack [2], based on differential cryptanalysis, has a different flavor. Differential cryptanalysis is a technique to analyze block ciphers. Essentially, differential cryptanalysis follows a bit slice through the

block cipher being analyzed, to characterize how it gets diffused. The goal of differential cryptanalysis is to identify bits leading to unusually high or low levels of diffusion. When such bits are identified, they can be used to recover bits of the encryption key. This can dramatically shrink the size of the key space, making brute force search realistic. As an example, differential cryptanalysis reduced the cost of key recovery attacks against the DES cipher from 256 encryptions to about 241.

Wang showed that a differential attack could produce collision in message digests, thereby breaking the collision-resistance of the hash function producing them. Wang first demonstrated her attack against MD4, MD5, RIPE-MD, and SHA-0. This was viewed as a stunning result, but then she demonstrated that a collision can be produced in SHA-1 at a cost of about 261 operations. This caused upheaval in the cryptographic community, raising the question as to whether we even understand what a hash function is.

The cryptographic community has vigorously debated hash design principles in the intervening years. The only clear consensus emerging from this debate is that we need a worldwide, focused project whose goal is to create a new generation of hash functions that defend against the new attacks. A lesson previously learned by the community is that contests have great efficacy in galvanizing technical consensus building. In 2007, NIST initiated an international competition to create a new hash standard [10]. Candidate submissions were due on October 31, 2008. 55 algorithms were entered. In February of this year, NIST whittled down the list of candidates to 40, and from this it plans to select 10 to 15 first-round candidates by August 2009. NIST plans to select a set of finalist algorithms in 2010, then to announce the winner(s) in 2011.

NIST is widely influential in the creation of cryptographic standards worldwide, so it is a good sponsor for the competition. One of NIST's

IN THIS ISSUE

- [Guest Editorial >>](#)
- [Stack Overflow >>](#)
- [CryptographicHash >>](#)
- [Letters>>](#)
- [Links>>](#)
- [Table of Contents >>](#)

most important contributions to cryptography standards has been the creation of requirements for algorithms submitted to the competition. The competition requires that candidate algorithms provide the collision-resistance, pre-image-resistance, and 2nd-pre-image-resistance properties — and be free of any known intellectual property. Algorithms must support output block sizes of 128, 160, 224, 256, 384, and 512 bits. The rules encourage support for features outside the core properties, especially for parallelization. Submissions must be accompanied by a security rationale, to help establish confidence in the algorithms.

Some New Designs

Two of the candidates submitted to the NIST hash competition, Skein and Vortex, include contributions by Intel personnel. Both are among the 40 entries remaining in the competition.

Skein

Skein was designed by Mihir Bellare (University of California, San Diego), Jon Callas (PGP Software), Niels Ferguson (Microsoft), Tadayoshi Kohno (University of Washington), Stefan Lucks (Bauhaus University-Mannheim), Bruce Schneier (British Telecom), Doug Whiting (Hi-Fn), and Jesse Walker (Intel Corporation). Skein produces message digests of any length from 1 to 296 bytes. Skein has three major components: a new block cipher named Threefish, a replacement for the cascade construction named Unique Block Iteration (UBI), and an argument system extending Skein’s domain of use beyond hashing.

The First Skein Component: The Threefish Block Cipher

Threefish is a “tweakable” block cipher [11], which means that a ran-

domizer called a tweak is passed to the cipher with the key and data to encrypt. Skein uses the block offset from the start of the message as the Threefish tweak. The tweak addresses many deficiencies in the cascade construction and represents the major innovation in Skein.

Threefish has three flavors: a 256-bit, a 512-bit, and a 1024-bit block size. The Threefish encryption key is the same size as the block size. The tweak is always 128 bits.

Threefish is a product cipher, meaning it is composed of rounds. Each round is a simple but weak encryption function. Threefish obtains security by piling round upon round: 72 rounds for Threefish-256 and for Threefish-512 and 80 rounds for Threefish-1024. The number of rounds represents a tradeoff between performance and security.

A Threefish round consists of a number of parallel MIX functions followed by a permutation, so that different blocks are mixed for different rounds. The MIX functions are made up of just three instructions — 64 bit addition, left rotate, and XOR — to combine two 64-bit words A and B, as depicted in Figure 1.

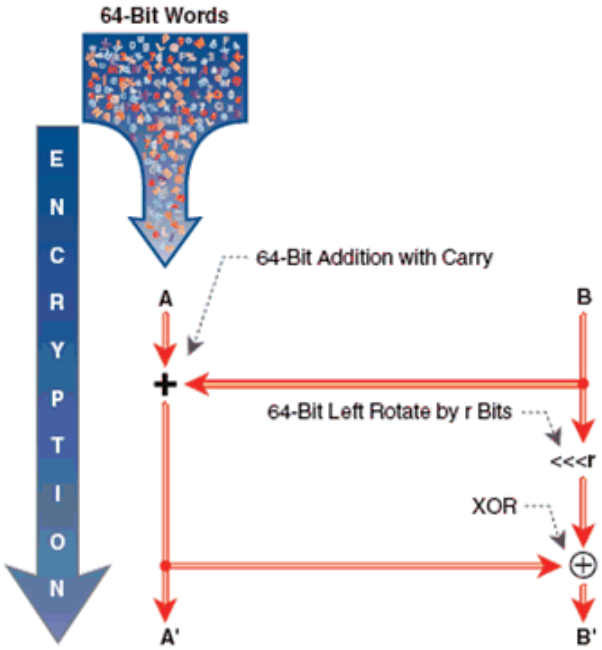


Figure 1: Threefish MIX (Source: Intel Corporation, 2009).

IN THIS ISSUE

- [Guest Editorial >>](#)
- [Stack Overflow >>](#)
- [Cryptographic Hash >>](#)
- [Letters >>](#)
- [Links >>](#)
- [Table of Contents >>](#)

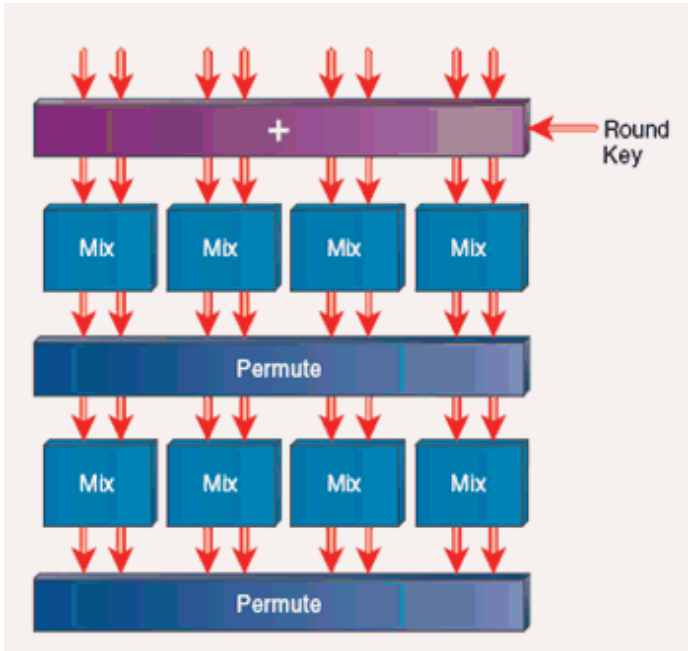


Figure 2: Two Rounds of Threefish-512 (Source: Intel Corporation, 2009).

efficiently exploit the super-scalar properties of modern processors. The rotation constants r were selected by a hill-climbing algorithm that maximized diffusion over randomly selected sets of rotation constants.

Threefish adds a round key every four rounds. Figure 2 depicts one of these additions. The Threefish round keys come from a key schedule inspired by Skipjack’s key schedule [12]. Each Threefish round key is the same size as the plaintext data block, and each key depends on all the bits of both the encryption key and the tweak.

The Second Skein Component: Unique Block Iteration

Unique Block Iteration (UBI) mode replaces the cascade construction in Skein. UBI consists of four parts. First, UBI uses the Matyas-Meyer-

Threefish-256 splits its input into four 64-bit words, so each round consists of two parallel MIXes: Threefish-512 uses eight words with four parallel MIXes, and Threefish-1024 uses sixteen words with eight parallel MIXes. Figure 2 depicts the Threefish round structure for Threefish-512. The parallel MIXes effi-

Oseas construction, $(iv, s) \rightarrow E_{iv}(s) \oplus s$, to build a compression function c out of any block cipher. Second, UBI padding appends enough 0 bits to bring the length of the message being hashed to a multiple of the block size. Third, UBI constructs and passes the tweak to the block cipher. The UBI tweak is composed of two flags and of the message block offset from the beginning of the message in bytes. One of the flags is set on the first block, and the second flag is set for the final block. Finally, UBI computes its output just like the cascade construction, the only difference being the construction of the tweak:

```
UBI( $iv, s$ )
pad( $s$ )
 $t \leftarrow 0 \oplus \text{start-flag}; iv_2 \leftarrow c(iv, s_1, t); t \leftarrow \text{block-size};$ 
do  $i = 2$  to  $b-1 \Rightarrow iv_{i+1} \leftarrow c(iv_i, s_i, t); t \leftarrow t + \text{block-size}$  od;
 $t \leftarrow t \oplus \text{end-flag}; \text{output } c(iv_b, s_b, t)$ 
```

UBI uses the Matyas-Meyer-Oseas construction instead of Davies-Meyer. This converts attacks against a hash function from related key attacks to chosen plaintext attacks against the block cipher: the community understands more about defending against the latter than it does about defending against the former.

The Third Skein Component: Skein Argument System

The Skein argument system extends the algorithm beyond normal hashing to application-specific or personalized hashing, message authentication, key derivation, pseudo-random number generation, stream ciphers, and tree (that is, parallelized) hashing.

Putting it Together in Skein

Skein instantiates UBI mode with the Threefish block cipher. The de-

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

sign's initialization vector is computed as the UBI-Threefish output of the configuration string "SHA-3." Skein first hashes a string s under UBI mode and iv to obtain an intermediate value. Skein uses the intermediate value as an iv to hash the integers $0, 1, 2, \dots$ under UBI, again to obtain the final output. Classical theory justifies the claim that Skein- n ($n = 256, 512, \text{ or } 1024$) achieves $n/2$ bits of security against collisions, and $n-1$ bits of security against 1st and 2nd pre-image attacks — the best that can be achieved, theoretically. The double hashing under UBI mode also allows Skein to make additional, unusually strong claims, as follows:

- If Threefish is a pseudo-random permutation, then Skein can be used as a pseudo-random function, a secure key derivation function, a secure message authentication code, a secure stream cipher, and a secure pseudo-random number generator.
- If Threefish acts like an ideal cipher, then Skein cannot be differentiated from a random oracle.

The first claim says that Skein can be used naively in a broad range of applications that usually require great sophistication when constructed from classical hash functions. The second claim is a non-trivial result: it claims that Skein is structurally sound when Threefish is viewed as a black box; that is, the attacker is not allowed to utilize any knowledge about the internals of Threefish. This structural property means the security of Skein depends on the security of the underlying block cipher only. The best known attack against Threefish at this time breaks a 34-round variant (out of 72 rounds for full Threefish), which is superior to AES, whose 8-out of 10-round variant falls to attack.

In software Skein is one of the fastest unbroken algorithms ever de-

vised: it runs at 6.1 clocks/byte on an Intel Core Duo processor and requires no special hardware acceleration, such as an AES round instruction. This is twice as fast as the best software implementations of the current hashing standard. Skein also maintains a very small footprint for its in-memory state, allowing implementation in even constrained environments such as smart cards.

Vortex

Vortex is a family of hash functions developed by Michael Kounavis and Shay Gueron of Intel. A main strength of the Vortex design is that this hash function can achieve an ideal performance of 2.2-2.5 cycles per byte by using the AES round [14] and carry-less multiply instructions [15]. Such instructions have been announced for future Intel processors. Vortex is one of the fastest collision-resistant hashes known when running on future IA processors, outperforming SHA-1 (approx. 7 cycles/byte) by 3.18X, and outperforming SHA256 (approx. 19 cycles/byte) by 8.63X.

The Vortex family produces message digests of 224, 256, 384, and 512 bits, respectively. The main idea behind Vortex is to use well-known algorithms with very fast diffusion in a small number of steps. These algorithms also balance the cryptographic strength, that comes from iterating block cipher rounds with S-box substitution and diffusion, against the need to have a lightweight implementation with as small a number of rounds as possible. Vortex is built upon the following algorithms:

- The Rijndael round function, which performs very fast mixing across 32 bits, as a standalone operation, and 128 bits or 256 bits, if combined with at least one more round.

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

- A variant of Galois Field multiplication that mixes bits of different sets in a manner that is cryptographically stronger than many other simpler schemes.

Vortex uses a variable number of Rijndael rounds with a stronger key schedule. The number of rounds is a tunable parameter. Rijndael rounds are followed by a variant of Galois Field multiplication to cross-mix between 128-bit or 256-bit sets. This transformation is not simple carry-less multiplication; rather, it combines bit reordering operations, XORs, and additions with carries. In this way, this variant of Galois Field multiplication achieves better diffusion than the straightforward carry-less multiplication between the 128-bit or 256-bit inputs; it is also a non-commutative operation, protecting against chaining variable swapping attacks.

Vortex uses the Enveloped Merkle-Damgård (EMD) construction to lift collision resistance, pre-image and 2nd pre-image resistance, pseudo-random oracle preservation, and pseudo-random function preservation from the underlying compression function to the hash function. To achieve its properties, the EMD construction first hashes the input string s under one initialization vector to get an intermediate value, and then it hashes the intermediate value under a second initialization vector to obtain a final result.

For Vortex-256, Gueron and Kounavis demonstrate that the number of queries required to find a collision with a probability greater or equal to 0.5 is at least $1.18 * 2^{122.55}$ [13].

In summary, the Vortex compression function uses the Rijndael round function. Vortex-224 and Vortex-256 use Rijndael-128 rounds.

Vortex-384 and Vortex-512 use Rijndael-256 rounds. AES uses the Rijndael-128 round function. For the remainder of this section, $\tilde{A}K(X)$ denotes a block cipher based on the Rijndael round function that encrypts X by using key K . $V_M^{(A)}(A, B)$ is a multiplication-based merging function.

The Vortex-block algorithm is the Vortex compression function. This algorithm incorporates two repetitions of an algorithm called Vortex-sub-block. The first repetition of Vortex-sub-block accepts as input the chaining variable $A_i || B_i$ and two least-significant input block words W_{4i}, W_{4i+1} of the message being hashed. It returns an intermediate value for the chaining variable $A || B$. The second repetition of Vortex-sub-block accepts as input the intermediate value of the chaining variable $A || B$ and two most-significant input block words W_{4i+2}, W_{4i+3} . It returns an update on the chaining variable $A_{i+1} || B_{i+1}$.

With the exception of the last sub-block (discussed later), the algorithm for processing a Vortex-sub-block is as follows:

Vortex sub-block (A, B, W_0, W_1)

// W_0 is the first word of the current sub-block to be processed

$A \leftarrow \tilde{A}_A(W_0) \oplus W_0; B \leftarrow \tilde{A}_B(W_0) \oplus W_0; A || B \leftarrow V_M^{(A)}(A, B)$

// W_1 is the second word of the current sub-block to be processed

$A \leftarrow \tilde{A}_A(W_1) \oplus W_1; B \leftarrow \tilde{A}_B(W_1) \oplus W_1; A || B \leftarrow V_M^{(A)}(A, B)$

output $A || B$

IN THIS ISSUE

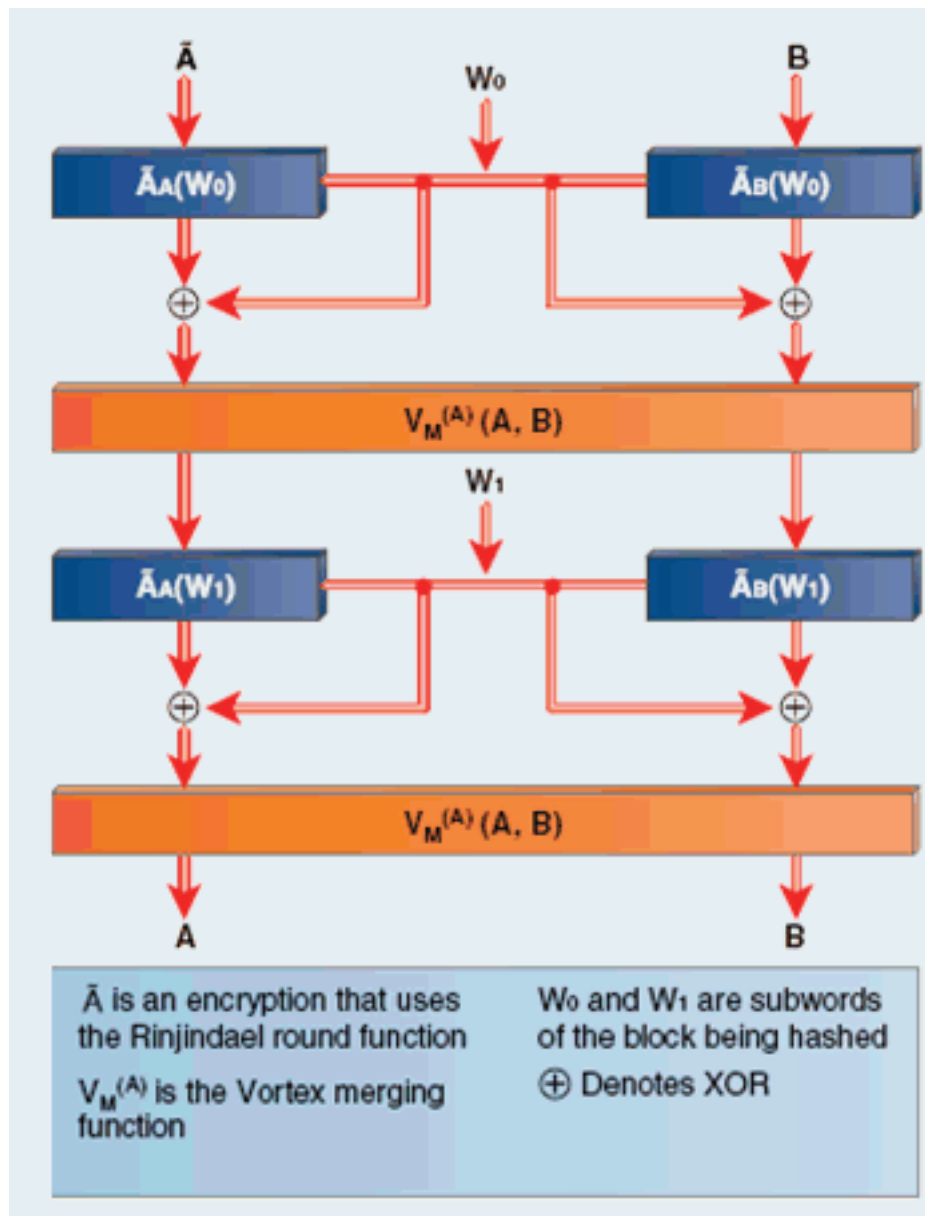
[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

Figure 3: Vortex Sub-Block Structure (Source: Intel Corporation, 2009).

The structure of the Vortex sub-block is shown in Figure 3. There are four instances of the transformation $\tilde{A}_K(x)$ in the Vortex sub-block. Each instance is wrapped by using a feed-forward provided by the Matyas-Meyer-Oseas construction to make the transformation non-reversible. The first two instances process input word w_0 . The other two instances process the input word w_1 . w_0 is the least-significant word of the current sub-block to be processed. Instances of $\tilde{A}_K(x)$ that accept the same input word process a different variable from among A , B . Each instance treats its input variable A or B as a key and treats its input word, which is one from w_0 or w_1 as plaintext, as that is the norm in a Matyas-Meyer-Oseas construction.

The Vortex merging function $V_M^{(A)}(A, B)$ operates as follows:

$$\begin{aligned} V_M^{(A)}(A, B) \\ A_1 A_0 \leftarrow A; B_1 B_0 \leftarrow B \\ O \leftarrow A_0 \otimes B_1; I \leftarrow A_1 \otimes B_0 \\ I_1 I_0 \leftarrow I; O_1 O_0 \leftarrow O \\ \text{output } B_1 \boxplus I_1 \parallel B_0 \boxplus O_0 \parallel A_1 \oplus O_1 \parallel A_0 \oplus I_0 \end{aligned}$$

where $\boxplus$ is ordinary 64-bit addition, and $\otimes$ denotes carry-less multiplication.

The Vortex merging function (as shown in Figure 4) ensures that the bits of A impact the bits of B and vice versa. In fact, each bit of one variable affects a significant number of the bits of the other variable in a non-linear manner. This makes the design better than a straightforward XOR or other simple mathematical operation. Carry-less multiplication is the default configuration of Vortex. The reason why Vortex uses carry-less multiplication by default is because it is easier to make analytical assertions about the collision resistance and pre-image

IN THIS ISSUE

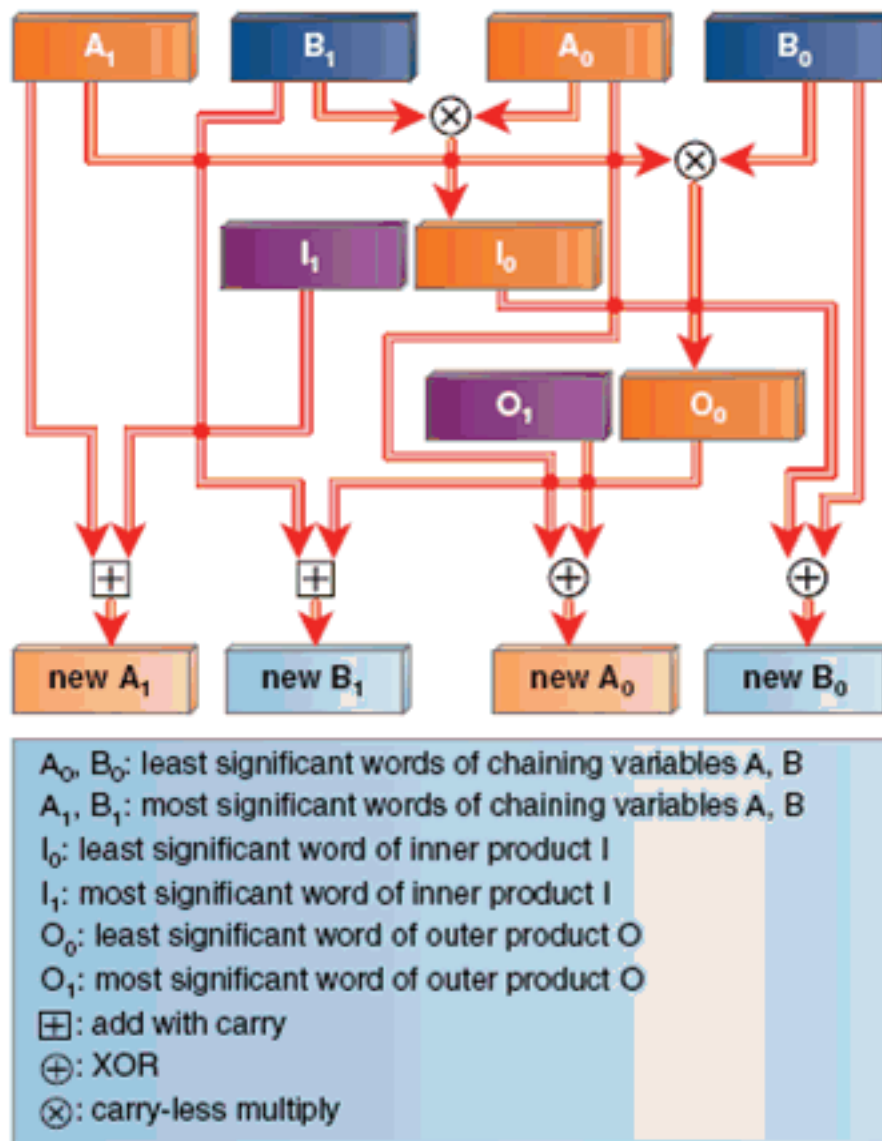
[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

Figure 4: Vortex Merging Function (Source: Intel Corporation, 2009).

resistance of the hash. In another configuration, Vortex uses integer multiplication. An integer multiplier increases the performance of the hash (not all processor architectures have a carry-less multiplier) and also increases the non-linearity of merging; however, it also makes the security of the scheme more difficult to prove.

The last Vortex sub-block is different. It repeats the sequence of Matyas-Meyer-Oseas transforms and merging several times. The total number of times every bit is diffused over all bits of the hash is determined by the number of sequences of Rijndael rounds and merging found in the last Vortex sub-block; this is another tunable parameter of the hash.

Design Rationale and Comparison

In this section, we summarize and compare the design rationale for both algorithms.

Skein Design Rationale

The Skein team had a number of goals:

- **Design for simplicity.** Simple designs, rather than complex ones, are easier to optimize and analyze for security flaws.
- **Maximize security per clock.** Performance trumps security in practice, so extract as much security as possible from each cycle. In particular, it is important to use only the simplest instructions that have been highly optimized on every platform. Exploit the super-scalar behavior of modern processors and maximize the diffusion from each operation. Through a series of experiments the Skein team discovered that simpler round functions with more rounds optimize both performance and security.

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

- **Achieve high performance and easy implementation on all processors.** Building in a performance advantage for one processor family over another is a disadvantage in a public competition.

The Skein team made a number of critical design decisions when designing the algorithm:

- **Base the design on a block cipher.** This is the most conservative security choice. The community knows better how to analyze designs based on block ciphers. Moreover, there is a well-established theory on how to turn a block cipher into a hash function.
- **Build your own block cipher.** The Skein team obtained better performance and a simpler security analysis with a new cipher of the correct block size instead of adapting an existing narrow-block cipher.
- **Make the block cipher tweakable.** This introduces great flexibility in designing a new mode at no cost.
- **Replace the cascade construction with something better.** The cascade construction has many known defects; the tweak allows UBI to offer provable security, and it suffers from none of the cascade construction's problems.
- **Use Matyas-Meyer-Oseas construction instead of Davies-Meyer.** Attacks against Matyas-Meyer-Oseas-based compression functions are chosen plaintext attacks; attacks against Davies-Meyer-based compression functions are related key attacks. The community has more experience defeating chosen plaintext attacks than related key attacks.

- **Do not use table lookups.** Most cipher designs use a table called an S-box. Lookups in the S-box enable side-channel attacks on software implementations, where the encryption key can be read by monitoring the power, timing, or EMI of the processor. Threefish is not subject to these attacks since it incorporates no table lookups.
- **Build in three different internal state sizes and allow output of any size.** This allows flexibility in the level of security available across different use cases.
- **Change the design when necessary.** The design should be changed when doing so allows for simpler security proofs.

Vortex Design Rationale

The Vortex team had a number of goals:

- **Design for performance. Vortex uses algorithms that are implemented by using dedicated instructions in future IA processors.** These are instructions for AES round computation (AES-NI) and carry-less multiplication (GFMUL-NI). The Vortex team argues that such instructions will become a trend in the industry.
- **Maximize security per clock. Like Skein, Vortex extracts as much security as possible from each cycle.** In particular, it uses independent AES round operations in each block. Such operations can potentially be completed in a single clock in future processors. It also introduces parallelism in the design of its compression function (two AES rounds and their key schedules can be executed in parallel); and finally, it maximizes the diffusion from each operation. Diffusion is supported by the AES round

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

algorithms (S-box substitution, ShiftRows, MixColumns) as well as by the multiplication stage that follows.

- **Achieve high performance and easy implementation on future processors.** The Vortex team believes that instructions for AES round computation and carry-less multiplication will become a trend in the industry. This is because (i) several hardware vendors including IBM and Sun are either implementing or researching them; (ii) even processors for embedded systems now include AES hardware; and (iii) there is a precedence in the industry that good instruction sets are widely adopted (for example, SSE instructions).

The Vortex team made a number of critical design decisions when designing the algorithm:

- **Base the design on a well-known and secure block cipher.** Vortex uses an AES round as a building block. AES is a well-studied block cipher, and the AES round operation offers very good mixing across 32 bits, as a standalone operation, and 128 bits if repeated several times.
- **Strengthen the AES key schedule.** Hashing is a one-way operation so additions with carries are permitted in the design of a hash function. Vortex strengthens the AES key schedule by adding round constants with carries and performing S-box substitution across all round key bytes.
- **Combine the outputs of two parallel AES transformations by using carry-less or integer multiplication.** Multiplication is a highly non-linear operation and can be used for destroying bit differentials.

- **Replace the cascade construction with something better.** Vortex uses the EMD construction and a tweak value to preserve the pseudo-random function and the pseudo-random oracle properties.
- **Use Matyas-Meyer-Oseas construction instead of Davies-Meyer.** This is similar to the rationale of the Skein team. The community has more experience defeating chosen plaintext attacks than related key attacks.
- **Use Rijndael S-boxes instead of table lookups.** Rijndael S-boxes have a special structure that allows them to be implemented by using combinatorial logic as opposed to table lookups. Thus, side-channel attacks can be averted.

Comparison of the Skein and Vortex Designs

It is now possible to highlight some similarities and differences between the Skein and Vortex designs. We begin with the similarities.

Similarities Between the Skein and Vortex Designs

Both designs are based on block ciphers, and they both use the Matyas-Meyer-Oseas construction to convert the related key attacks against the deployed hash function designs into chosen plaintext attacks. The cryptographic community has more experience defeating chosen plaintext attacks than related key attacks.

Both designs use a flavor of the cascade construction to paste together the hash of different blocks output by a compression function, and they both double hash the final output; that is, the output from the cascade construction is rehashed to become the final output. Both do this to address vulnerabilities that arise from processing the final block

IN THIS ISSUE[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

with a construction such as Davies-Meyer or Matyas-Meyer-Oseas, within the cascade construction.

Differences Between the Skein and Vortex Designs

The differing design approaches reflect the differing skill sets of the two teams. The Skein team members were skilled in designing block ciphers. In contrast, the Vortex team members did not design a new block cipher, but instead used an existing one. The Skein design allows its security claims to be derived from the security of a block cipher. The Vortex design effort emerged from the need to demonstrate a secure hash function by using the new AES round and carry-less multiplication instructions for future IA processors, which the Vortex team members designed.

Skein uses significantly more rounds (72/72/80) than Vortex, stemming from the different design decisions made by each team. The Skein team's experiments indicated diffusion per clock is maximized by numerous simple rounds. The Vortex designers instead were motivated by performance; they believed the best performance is achieved by using fewer rounds, based on very powerful diffusion primitives.

Skein's support for a hash value of any length from 1 to 264 bytes allowed the team to prove the property that Skein cannot be differentiated from a random oracle if Threefish acts like an ideal cipher. There are two ways to think about any algorithm: (1) as a monolithic black box, where you have no knowledge of any of the algorithm's internal parts, and (2) as presented in this article, where we know the details of the algorithm's internal structure. By saying that Skein cannot be differentiated from a random oracle, therefore, we mean that it is impossible, even in principle, for Skein to construct any statistical test that exploits differences in the two views. This means that Skein is struc-

turally sound and that its security depends only on the security of Threefish. Vortex's final output is of a fixed length, but the second hash allows it to act like a fixed-length random oracle.

None of the Skein components use table lookups such as S-boxes, so it is more difficult to launch side-channel attacks on Skein than on specific implementations of Vortex. Vortex avoids these attacks by relying on a hardware logic implementation for the AES S-boxes.

Summary

In this article we reviewed the theory of hash functions, the state of knowledge about them, and some of Intel's contributions to this field of research. Hash functions are basic building blocks that are central to cryptography's mission, but recent attacks by Joux and Wang have undermined our confidence in classical constructions. Because of this insecurity, there has been a wave of new research into hash functions, and Intel has been at the forefront, with two independent and radically different submissions to the international NIST hash competition.

References

- [1] A. Joux. "Iterated Collisions on Iterated Hash Functions." *Crypto 2004, Lecture Notes in Computer Science (LNCS) 3621*, Springer-Verlag, Berlin, 2005.
- [2] X. Wang, Y. Yin, and H. Yu. "Finding Collisions in the full SHA-1." *Crypto 2005, LNCS 2947*, Springer-Verlag, Berlin, 2004
- [3] A. Lenstra, X. Wang, and B. de Berger. "Colliding X.509 Certificates." <http://eprint.iacr.org>
- [4] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
- [5] R. Merkle. "Secrecy, authentication, and public key systems." Stan-

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

ford University Ph.D. thesis, 1979. Available as "Technical Report No. 1979-1," Information Systems Laboratory, Stanford University, Palo Alto, California, 1979.

[6] I. Damgård. "A Design Principle for Hash Functions." *Crypto 1989*, LNCS 435, Springer-Verlag, Berlin, 1989.

[7] R. Winternitz. "A secure one-way hash function built from DES." In *Proceedings of IEEE Symposium on Security and Privacy*, 1984.

[8] S. Matyas, C. Meyer, and J. Oseas. "Generating strong one-way functions with cryptographic algorithms." *IBM Technical Bulletin*, 27, 1985.

[9] J. Black, P. Rogaway, and T. Shrimpton. "Black box analysis of block-cipher-based hash functions from PGV." *Crypto 2002*, LNCS 2442, Springer-Verlag, Berlin, 2002.

[10] The National Institute of Standards and Technology. US Government agency sponsor for competition on next-generation hash design. <http://csrc.nist.gov>

[11] M. Liskov, R. Rivest, and D. Wagner. "Tweakable Block Ciphers." *Crypto*, LNCS 2442, Springer-Verlag, Berlin, 2002.

[12] National Security Agency. Skipjack and KEA Specifications. May 29, 1998. <http://csrc.nist.gov>

[13] M. Kounavis and S. Gueron. "Vortex: A new Family of One-Way Hash Functions Based on Rijndael Rounds and Carry-less Multiplication." <http://eprint.iacr.org>

[14] S. Gueron. "Advanced Encryption Standard (AES) Instructions Set." <http://software.intel.com>

[15] S. Gueron and M. Kounavis. "Carry-Less Multiplication and its Usage for Computing the GCM Mode." <http://software.intel.com>

— *Jesse Walker is an Applied Cryptographer in Intel's Communication Technology Laboratory. He can be contacted at jesse.walker@intel.com. Michael E Kounavis is a*

Senior Research Scientist working with Intel Labs. His e-mail is michael.e.kounavis@intel.com. Shay Gueron is an Intel Principal Engineer. He can be reached at shay.gueron@intel.com. Gary Graunke is a Senior Staff Architect for cryptography at Intel. His e-mail is gary.graunke@intel.com. Copyright (c) 2009 Intel Corporation. All rights reserved. This article and more on similar subjects may be found in the Intel Technology Journal, June 2009 Edition, Advances in Internet Security.

More hashing algorithm exploration is provided in a *Dr. Dobb's* series authored by José R.C.Cruz in late 2011 exploring the new cryptography standard candidates.

"Finding the New Encryption Standard, SHA-3" (<http://www.drdoobs.com/security/231700137>) details the search for a replacement for SHA-2, focusing on five finalists, their internals, and performance.

"Testing the Final SHA-3 Hashing Algorithms" (<http://www.drdoobs.com/security/231900574>) focuses on testing the finalists in the competition for a new SHA-3 standard and shows generally fast, secure hashing algorithms with few collisions.

[Comment](#)

IN THIS ISSUE[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

This Month on DrDobbs.com

Items of special interest posted on www.drdobbs.com over the past month that you may have missed

THE DESIGN OF LLVM

In the last five years, LLVM has evolved from an academic project to the universal back-end of C, C++, and Objective C compilers. The key to its success is its performance and adaptability, both of which derive from its unique design and implementation.

<http://www.drdobbs.com/cpp/240001128>

THE PARROT ASTEROID DEVELOPMENT EXPERIENCE

Asteroid uses Bluetooth to pair with your phone for hands-free calling, and the dual-microphone accessory helps to improve the noise cancellation for crystal clear hands-free phone conversations.

<http://www.drdobbs.com/blogs/mobile/240001339>

THE LOCALSTORAGE API

To use the localStorage API, developers rely on a wide choice of libraries, each with their own strengths and weaknesses. Pamela Fox examines the major contenders and explains where their features can best be used.

<http://www.drdobbs.com/web-development/240000682>

AUTOMATIC REFERENCE COUNTING ON IOS

Automatic reference counting is an innovative way of managing Objective-C objects on MacOS X 10.7 and iOS 5. It does away with explicit retain, release and autorelease messages, and it behaves the same on both platforms.

<http://www.drdobbs.com/mobile/240000820>

WE WHO VALUE SIMPLICITY HAVE BUILT INCOMPREHENSIBLE MACHINES

How creeping feature additions turn good ideas into morasses of incomprehensible options.

<http://www.drdobbs.com/architecture-and-design/240000813>

EXTRACTING FUNCTION PARAMETER AND RETURN TYPES IN C++

Using the metaprogramming capabilities of C++ and generic programming, it's possible to create an elegant parser of parameter types. Command-line interpreters, parsers, and syntax checkers are among the many applications.

<http://www.drdobbs.com/cpp/240000586>

IN THIS ISSUE

- Guest Editorial >>
- Stack Overflow >>
- Cryptographic Hash >>
- Letters >>
- Links >>
- Table of Contents >>

Dr.Dobb's

Andrew Binstock Editor in Chief, Dr. Dobb's
alb@drdobbs.com

Deirdre Blake Managing Editor, Dr. Dobb's
dblake@techweb.com

J.D. Hildebrand Associate Editor, Dr. Dobb's
jdhildebrand@drdobbs.com

Amy Stephens Copyeditor, Dr. Dobb's
astephens@techweb.com

Sean Coady Webmaster, Dr. Dobb's
scoady@techweb.com

Jon Erickson Editor in Chief Emeritus , Dr. Dobb's

CONTRIBUTING EDITORS

Scott Ambler
Mike Riley
Herb Sutter

DR DOBB'S
UBM TECHWEB
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

INFORMATIONWEEK

Rob Preston VP and Editor In Chief, Information-Week
rpreston@techweb.com 516-562-5692

John Foley Editor, InformationWeek
jpfoley@techweb.com 516-562-7189

Chris Murphy Editor, InformationWeek
cjmurphy@techweb.com 414-906-5331

Art Wittmann VP and Director, Analytics, InformationWeek
awittmann@techweb.com 408-416-3227

Alexander Wolfe Editor In Chief, Information-Week.com
awolfe@techweb.com 516-562-7821

Stacey Peterson Executive Editor, Quality, InformationWeek
speterson@techweb.com 516-562-5933

Lorna Garey Executive Editor, Analytics, InformationWeek
lgarey@techweb.com 978-694-1681

Stephanie Stahl Executive Editor, Information-Week
sstahl@techweb.com 703-266-6030

Fritz Nelson VP and Editorial Director
fnelson@techweb.com 949-223-3608

David Berlind Chief Content Officer, TechWeb
dberlind@techweb.com 978-462-5315

REPORTERS

Charles Babcock Editor At Large
Open source, infrastructure, virtualization
cbabcock@techweb.com 415-947-6133

Thomas Claburn Editor At Large
Security, search, Web applications
tclaburn@techweb.com 415-947-6820

Paul McDougall Editor At Large
Software, IT services, outsourcing
pmcdougall@techweb.com

J. Nicholas Hoover Senior Editor
Desktop software, Enterprise 2.0,
collaboration
nhoover@techweb.com 516-562-5032

Andrew Conry-Murray New Products and Business Editor
Information and content management
acmurray@techweb.com 724-266-1310

W. David Gardner News Writer
Networking, telecom
wdavidg@earthlink.net

Antone Gonsalves News Writer
Processors, PCs, servers
antoneg@pacbell.net

Eric Zeman
Mobile and Wireless
eric@zemanmedia.com

CONTRIBUTORS

Michael Biddick mbiddick@nwc.com
Michael A. Davis mdavis@nwc.com
Jonathan Feldman jfeldman@nwc.com
Randy George rgeorge@nwc.com
Michael Healey mhealey@nwc.com

EDITORS

Jim Donahue Chief Copy Editor
jdonahue@techweb.com

ART/DESIGN

Mary Ellen Forte Senior Art Director
mforte@techweb.com

Sek Leung Senior Designer
sleung@techweb.com

INFORMATIONWEEK ANALYTICS
analytics.informationweek.com

Art Wittmann VP and Director
awittmann@techweb.com 408-416-3227

Lorna Garey Executive Editor, Analytics
lgarey@techweb.com 978-694-1681

Heather Vallis Managing Editor, Research
hvallis@techweb.com 508-416-1101

INFORMATIONWEEK.COM

Benjamin Tomkins Managing Editor
btomkins@techweb.com 516-562-5336

Roma Nowak Senior Director,
Online Operations and Production
rnowak@techweb.com 516-562-5274

Tom LaSusa Managing Editor,
Newsletters
tlasusa@techweb.com

Jeanette Hafke Web Production Manager
jhafke@techweb.com

Joy Culbertson Web Producer
jculbertson@techweb.com

Nevin Berger Senior Director,
User Experience
nberger@techweb.com

Steve Gilliard Senior Director,
Web Development
sgilliard@techweb.com

Copyright 2012 United Business
Media LLC. All rights reserved.



INFORMATIONWEEK
ADVISORY BOARD

Dave Bent
Senior VP and CIO
United Stationers

Robert Carter
Executive VP and CIO
FedEx

Michael Cuddy
VP and CIO
Toromont Industries

Laurie Douglas
Senior CIO
Publix Super Markets

Dan Drawbaugh
CIO
University of Pittsburgh
Medical Center

Jerry Johnson
CIO
Pacific Northwest National
Laboratory

Kent Kushar
VP and CIO
E.&J. Gallo Winery

Carolyn LEclipse CDTON
Director, E-Services
California Office of the CIO

Jason Julynard
Managing Director
Wells Fargo Securities

Randall Mott
Sr. Executive VP and CIO
Hewlett-Packard

Denis O'Leary
Former Executive VP
Chase.com

Mykolas Rambus
CEO
Wealth-X

M.R. Rangaswami
Founder
Sand Hill Group

Manjit Singh
CIO
Las Vegas Sands

David Smoley
CIO
Flextronics

Ralph J. Szygenda
Former Group VP and CIO
General Motors

Peter Whatnell
CIO
Sunoco

UBM TECHNOLOGY

Paul Miller CEO

John Dennehy, Chief Financial
Officer
jdennehy@techweb.com

David Michael, Chief Informa-
tion Officer michael@tech-
web.com

Scott Vaughan, Chief Market-
ing Officer svaughan@tech-
web.com

David Berlind, Chief Content
Officer dberlind@techweb.com

Harris Grayman, SVP, People &
Culture, UBM Technology har-
ris.grayman@ubm.com

Ed Grossman, EVP, Informa-
tionWeek Business Technology
Network egrossman@tech-
web.com

Martha Schwartz, EVP, Sales,
InformationWeek Business
Technology Network
mschwartz@techweb.com

Joseph Braue, EVP, Light Read-
ing Communications Network
jbraue@techweb.com

Simon Carless, EVP, UBM Tech-
Web Game Network
scarless@techweb.com

Lenny Heymann, EVP and
Group General Manager, UBM
TechWeb Events Network lhey-
mann@techweb.com

Marco Pardi, EVP, Sales, UBM
TechWeb Events Network
mpardi@techweb.com

Fritz Nelson, Vice President,
Editorial Director Information-
Week Business Technology Net-
work

John Ecke, VP of Brand and
Product Development, Informa-
tionWeek Business Technology
Network jecke@techweb.com

Fred Knight, GM and Co-Chair,
Enterprise Connect
fknight@techweb.com

Lori Silva, VP, UBM Events &
Operations lori.silva@ubm.com

Frank Sliwka, Vice
President/European Business
Development and Event Direc-
tor, GDC Europe fsliwka@tech-
web.com

UNITED BUSINESS MEDIA LLC

Pat Nohilly Sr. VP, Strategic De-
velopment
and Business Administration

Marie Myers Sr. VP,
Manufacturing

INFORMATIONWEEK VIDEO

informationweek.com/tv

Fritz Nelson Executive
Producer
fnelson@techweb.com

INFORMATIONWEEK
BUSINESS
TECHNOLOGY
NETWORK

DarkReading.com

Security
Tim Wilson, Site Editor
wilson@darkreading.com

IntelligentEnterprise.com
App Architecture
Doug Henschen,
Editor in Chief
dhenschen@techweb.com

NetworkComputing.com
Networking , Communications,
and Storage
Mike Fratto, Site Editor
mfratto@techweb.com

PlugIntoTheCloud.com
Cloud Computing
John Foley, Site Editor
jpfoley@techweb.com

InformationWeek SMB
Technology for Small
and Midsize Business
Benjamin Tomkins,
Site Editor
btomkins@techweb.com

Dr. Dobb's
Good Stuff for Serious
Developers
Andrew Binstock
Editor in Chief
alb@drdobbs.com

IN THIS ISSUE

[Guest Editorial >>](#)[Stack Overflow >>](#)[Cryptographic Hash >>](#)[Letters >>](#)[Links >>](#)[Table of Contents >>](#)

Dr.Dobb's Business Contacts

INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK

EVP of Group Sales,
InformationWeek Business Technology Network,
Martha Schwartz
 (212) 600-3015, mschwartz@techweb.com

Sales Assistant, Salvatore Silletti
 (212) 600-3327, ssilletti@techweb.com

SALES CONTACTS—WEST

Western U.S. (Pacific and Mountain states)
 and Western Canada (British Columbia,
 Alberta)

Sales Director, Michele Hurabiell
 (415) 378-3540, mhurabiell@techweb.com

Strategic Accounts

Account Director, Sandra Kupiec
 (415) 947-6922, skupiec@techweb.com

Account Manager, Vesna Beso
 (415) 947-6104, vbese@techweb.com

Account Executive, Matthew Cohen-Meyer
 (415) 947-6214, mmeyer@techweb.com

MARKETING

VP, Marketing, Winnie Ng-Schuchman
 (631) 406-6507, wng@techweb.com

Marketing Director, Angela Lee-Moll
 (516) 562-5803, aleemoll@techweb.com

Marketing Manager, Monique Kakegawa
 (949) 223-3609, mluttrell@techweb.com

Director, Client Marketing, Michelle Somers
 (516) 562-7928, msomers@techweb.com

SALES CONTACTS—EAST

Midwest, South, Northeast U.S. and Eastern Canada
 (Saskatchewan, Ontario, Quebec, New Brunswick)

District Manager, Steven Sorhaindo
 (212) 600-3092, ssorhaindo@techweb.com

Strategic Accounts

District Manager, Mary Hyland
 (516) 562-5120, mhyland@techweb.com

Account Manager, Tara Bradeen
 (212) 600-3387, tbradeen@techweb.com

Account Manager, Jennifer Gambino
 (516) 562-5651, jgambino@techweb.com

Account Manager, Elyse Cowen
 (212) 600-3051, ecowen@techweb.com

Sales Assistant, Kathleen Jurina
 (212) 600-3170, kjurina@techweb.com

AUDIENCE DEVELOPMENT

Director, Karen McAleer
 (516) 562-7833, kmcaleer@techweb.com

BUSINESS OFFICE

General Manager, Marian Dujmovits

United Business Media LLC
 600 Community Drive
 Manhasset, N.Y. 11030 (516) 562-5000
Copyright 2012. All rights reserved.

Entire contents Copyright © 2012, Techweb/United Business Media LLC, except where otherwise noted. No portion of this publication may be reproduced, stored, transmitted in any form, including computer retrieval, without written permission from the publisher. All Rights Reserved. Articles express the opinion of the author and are not necessarily the opinion of the publisher. Published by Techweb, United Business Media, 303 Second Street, Suite 900 South Tower, San Francisco, CA 94107 USA 415-947-6000.

UBM TECHWEB

Paul Miller CEO

John Dennehy CFO

David Michael CIO

Joseph Braue Sr.VP, Light Reading
 Communications Network

Scott Vaughan CMO

Ed Grossman Executive Vice President, Information-
 Week Business Technology Network

John Ecke VP and Group Publisher,
 Financial Technology Network, InformationWeek
 Government, InformationWeek Healthcare

Martha Schwartz EVP, Group Sales,
 InformationWeek Business Technology Network

Beth Rivera Senior VP, Human Resources

David Berlind Chief Content Officer,
 TechWeb, and Editor in Chief, TechWeb.com

Fritz Nelson VP, Editorial Director,
 InformationWeek Business Technology
 Network, and Executive Producer, TechWeb TV

Eric Lundquist VP and Editorial Analyst, Information-
 Week Business Technology Network

UNITED BUSINESS MEDIA LLC

Pat Nohilly Sr.VP, Strategic Development and Business
 Admin.

Marie Myers Sr.VP, Manufacturing

