Low-Threat Security Patches and Tools

Mohd A. Bashar[*] Ganesh Krishnan, Markus G. Kuhn,
Eugene H. Spafford, S. S. Wagstaff, Jr.

COAST Laboratory

Department of Computer Sciences
Purdue University
1398 Department of Computer Sciences
West Lafayette, IN 47907–1398

{krishg,kuhn,spaf,ssw}@cs.purdue.edu

30 November 1996

Abstract

We consider the problem of distributing potentially dangerous information to a number of competing parties. As a prime example, we focus on the issue of distributing security patches to software. These patches implicitly contain vulnerability information that may be abused to jeopardize the security of other systems. When a vendor supplies a binary program patch, different users may receive it at different times. The differential application times of the patch create a window of vulnerability until all users have installed the patch. An abuser might analyze the binary patch before others install it. Armed with this information, he might be able to abuse another user's machine.

A related situation occurs in the deployment of security tools. However, many tools will necessarily encode vulnerability information or explicit information about security "localisms." This information may be reverse-engineered and used against systems.

We discuss several ways in which security patches and tools may be made safer. Among these are: customizing patches to apply to only one machine, disguising patches to hinder their interpretation, synchronizing patch distribution to shrink the window of vulnerability, applying patches automatically, and using cryptoprocessors with

[*]Current address: Alamadanga Kushtia, Bangladesh.

enciphered operating systems. We conclude with some observations on the utility and effectiveness of these methods.

1 Introduction

1.1 The general problem

Suppose Zelda wishes to distribute sensitive information to Alice and Bob. There are several potential problems in the process that we know how to manage: preventing others from reading the information, preventing others from altering the information, and marking the information in such a way that Alice and Bob know who sent it. We know how to scale these solutions affordably for many situations. We also know how to configure the solutions to handle cases where Zelda sends frequent messages to different, but not necessarily disjoint, sets of users, e.g., message 1 to Alice, Bob and Carol; message 2 to Alice; and message 3 to Bob, Carol and David.

We have identified a class of situations to which there are as yet no formalized solutions. These situations occur when Zelda distributes information to Alice, Bob and others who may be potential rivals. The information offers each of them a competitive advantage if they receive and act on the information before one of the others. Examples include distributing financial market information to investors, and providing bidding specifications to potential contractors. Part of this problem is determining how to distribute and protect the information in such a way as to reduce or eliminate the time during which the difference in knowledge may be exploited. Another major part of the problem is how to scale any solution to large numbers of receivers, and how to accomplish this inexpensively.

Of particular interest to us are the cases of distributing security-relevant updates and patches to software. When a vendor distributes a security-related patch to customers, it contains implicit information about the vulnerability involved, and perhaps of the exploit itself. The patch must be sent to customers and users if the vulnerability is known to others. However, the nature of patch distribution is such that many users may not receive (or use) patch information at the same time as others. There are global differences in time zones, work weeks, holidays, workloads, and competence. During the time between the first receipt of the patch, and the application of that patch to the last remaining machine needing it may be a large window of vulnerability. Our concern is how to reduce this vulnerability, raise the cost of exploiting it, somehow "tag" the exploiter for later action, and otherwise make the process safer for all the recipients.

The remainder of this paper discusses aspects of the general set of problems in the context of vendor patch distribution. Although this does not have all the characteristics present in the general problem, it is one with which most people are familiar, and presents sufficient complexity and risk to warrant concern. In addition, we also discuss how some of our solutions may also be applied to a closely-related problem: that of protecting security tools developed or employed locally to each site. Each tool set contains an implicit list of vulnerabilities—especially if customized for local conditions and concerns—that may be exploited if the tools are obtained by another and analyzed. In fact, as noted in [12] and [6], the tools may be modified and then used as automated attack mechanisms. This represents a different aspect of the general problem: one where distribution may also occur to unauthorized parties of unknown number, and where the window of vulnerability may be arbitrarily large.

1.2 Summary of possible solutions

This paper investigates how patch distribution and security tool distribution can be made safer. We explore methods of protecting this information during distribution and employment, and discuss the limitations of any such protection. Although we suspect that it may be impossible to guarantee the complete safety of distributed vulnerability-related information, we demonstrate that there may be effective means of reducing the risk associated with such distributions.

The crux of the patch distribution problem is this: how are we to distribute the solution of a problem without betraying any information about the problem? This is difficult because the solution of a problem by its nature contains clues about the problem. Thus, it may well be that the patch distribution problem we consdier cannot be solved in its entirety. Therefore, we must also consider ways to reduce the associated risk.

In the following sections, we consider the following methods of reducing the risks accompanying security-relevent patch distribution:

- We can "customize" each patch or tool so that each one differs from machine to machine. This will result in software that is different for any two sites, possibly disguising common vulnerabilities. Furthermore, if the original software was likewise customized, then it is not clear that reverse-engineering one particular patch set or tool would result in generically useful information. This is, in some ways, related to the way polymorphic computer viruses alter their structure to avoid

detection by pattern matching (e.g.,. [15]).

- We can introduce "noise" to mask changes. If a sufficiently large number of other, non-functional changes are made to a system when a patch is installed, then it should be correspondingly difficult to determine the aspects of the patch that are critical to security.
- We can synchronize patch distribution and application so that all users receive and install the patch at the same time. Given the uncertainties of the present patch distribution channels and the varying degree of security-consciousness among users, the goal of achieving perfect synchrony in patch distribution and application seems impossible to attain. Thus, we need to consider "loose" synchronization methods, or prioritized distribution.
- We can use automated patching: Part of the operating system patches itself when it receives an authenticated command over the network from the vendor.
- Another approach involves cryptographic methods to obscure patches. Presumably, a patch could be enciphered during distribution to prevent the problems described earlier in this paper. This is not a simple approach, as it raises additional questions: how would it be deciphered for installation? And if it were installed somehow without the user seeing it, could he deduce its form by comparing the old and new operating systems? How would encryption protect the contents if the antagonist were a legitimate recipient of the patch and the associated keys? Could special hardware assist with encryption and decryption?
- A somewhat whimsical solution to the problem would be to have trusted guards carry the patch to each site and force the system managers install the patch at precisely noon GMT on a certain day.

In what follows, we classify solutions not requiring armed guards as either software or hardware solutions. The software solutions are expected to run on standard computer hardware. The hardware solutions require each computer to have special hardware or firmware. The solutions requiring deployment of armed guards , although appealing, is impractical for most present-day software: several vendors' products would require a full-time agent present at each site.

2 Software solutions

These solutions will run on ordinary computers except that the vendor's computer may require a good random number generator that might involve some special hardware (cf. [5]). Also, one of the solutions in section 2.3 uses time locks, which might use special hardware to solve a puzzle. But this requires only the machines used in the solution, and no user machines, to have special hardware.

2.1 Customization

Each site or machine has its own unique Operating System (OS) binary code. The vendor's compiler uses a Good Random Number Generator (GRNG) to determine code arrangement, register assignment, variable assignment, etc.[1] The vendor saves the sequence from the GRNG used for each site so that it can prepare a patch that applies only to that one particular site.

As different sites have slightly different OS's, they might have different flaws and require patches for different problems. Thus, if a malicious user looks at the patch or compares the old and new binaries to learn what problem the patch fixes, then she might not be able to use this knowledge to break into any other systems because perhaps only her system had that bug. However, some OS bugs (design errors) may have such general nature that they apply to all (or many) versions of the OS regardless of the use of the GRNG when it was compiled. Then a malicious user could harm systems that installed the patch later. These random variations in code for a given program are used also in section 2.2 Obfuscation below.

2.2 Obfuscation

The patch is disguised, but not enciphered, to hinder, but not completely prevent, reverse engineering.

As in section 2.1, Customization, the vendor's compiler uses a GRNG to determine code arrangement, make register assignments, and other changes. The GRNG could also be used to introduce complex boolean expressions by expanding the parse tree in those portions of the OS that are not time-critical. These changes would make the code much more difficult for the attacker to analyze, or possibly render the code impossible to understand.

[1] A pseudo-random generator is not appropriate, as discovery of the generator may allow an attacker to reproduce the sequence of perturbations in the compilation. This comment applies to the other schemes were we describe use of a GRNG.

Indeed, optimization itself may provide sufficient obfuscation of the program. In contrast to section 2.1, now each site has the same version of the OS generated by the same sequence from the GRNG. When the vendor fixes the flaw, he recompiles the OS using a new sequence for the GRNG. The malicious user who compares the old and new binary files will find thousands (or more) of differences and thus have great difficulty discovering the security flaw.

In a slight variation of this idea, the changes are drawn from a database of harmless variations of the compiled code constructed when the OS was compiled. (In later sections, we describe how to construct such a database of false changes.) Almost all of these modifications are composed of semantically equivalent changes of register assignment or order of execution of commutative operations (e.g., `b+a` instead of `a+b`). Only a few, and possible no, changes in a set repair a real security problem. The malicious user examining the set of changes would have to expend considerable effort each month to find a security fix, and some months she would find nothing.

In this scheme, the vendor distributes a set of "false changes" at a regular interval—say, 100 false changes per month—along with any real change that may have been created during the intervening period. A false change is effectively a placebo that leaves the functional behavior of a program unchanged. These false changes act as noise that makes the real fix less discernible. False changes must be generated under the following constraints:

1. The functional behavior of the program must remain unchanged.
2. The program execution time should not suffer.
3. The additional space requirement should not be too high.
4. The change code should blend well with the existing program code; that is, it should be difficult to distinguish real changes from false ones, otherwise an abuser will see through them for what they really are.

As a variation of issuing 100 changes per month, the vendor might issue sets of 100 changes at random times satisfying a Poisson distribution with mean one month. That way if it were necessary to repair a serious security hole quickly, no one would notice the extra issue because it might have occurred then anyway.

Some managers might install all changes; some managers might miss some. How will the vendor debug systems with so many different versions of the OS in use? The answer is the standard one: The manager must get his system up-to-date first.

2.3 Synchronized patch installation

We assume that all the computers are on networks and each network is connected to some site which in turn is connected to the a common network (e.g., a dedicated private network, or the Internet). A site is under a single administrative control and may contain multiple networks. In one variation, each site has a security class as well. Higher security classes are assigned to sites with greater need for protection and smaller chance of having malicious users. Every site has a locally-trusted machine designated as the local patch distributor through which encrypted patches and keys are distributed to the local computers.

On the next higher level in the distribution hierarchy there is a set of machines designated as regional distributors, each of which connects logically to the set of local distributors. The regional distributors, along with a root distributor, may be maintained by a vendor, a cartel of vendors, or some independent body serving the industry.[2]

When a new patch is issued, the root distributor produces several encrypted versions of it using different keys—one key for each security class—and sends the encrypted patches and keys to all the regional distributors. Regional distributors then send the appropriate version of the encrypted patches to all local distributors under their respective domains, and the local distributors forward it to all machines within their respective sites. Having distributed the encrypted patch, the regional distributors coordinate among themselves to ensure that all sites with high security class have received the patch. Then the regional distributors give out the keys to the local distributors in successive waves—sites with the highest security class receive their keys first and those with the lowest security class receive it last. The regional distributors may again coordinate among themselves to ensure that all higher security sites have received the keys before distributing keys to the lower security sites.

The above scheme does not work for a machine that is either switched off or temporarily disconnected from the network when the patch and the key are distributed. To correct the situation, when this machine boots up or reconnects back to network and before it executes any other process, it contacts the local distributor and receives any patch that might have been issued during the intervening period.

In a variation of this approach the patch is enciphered and sent to all sites or made available by `ftp` from the vendor. With it are included (in

[2]We should note that this loosely corresponds to the current logical organization of FIRST response teams.

plain text) instructions to install it at noon GMT on a certain day, at which time the key to the cipher will be revealed. Cliff Stoll first described this scenerio[17] and suggested that one good way to reveal the key would be to publish it in a national newspaper such as *USA Today* or the *New York Times*.

Other methods of distributing the key would be for the vendor to place itin a public directory at noon GMT so that all sites may `ftp` it at the same time; the vendor could email the key to all sites at once; or a special-purpose protocol could be developed to broadcast the key to the whole network. Any of these solutions would work if the number of sites was small, if they were all on the same network, and enough of the machines on the network were working at noon GMT so that all sites could get the key and install the patch within a few minutes. If there were many sites to be patched or if they were located on more than one network, the scale of distribution renders these schemes inefficient or impossible. In that case, one might use some form of time locks (first suggested in [16] and independently developed in [13]) to reveal the key (or keys) at the same time in different places.

One approach to time locks is to have each time lock server solve an inherently sequential "time lock puzzle" which requires a precise amount of computing to solve, and whose solution is the key. This sort of time lock puzzle probably would not be suitable because some computers are much faster than others and a close approximation to synchrony is important in patch distribution. For example, one might be a PC and another a Cruncher [4], which can solve "time lock puzzles" requiring arithmetic with large integers hundreds of times faster than a PC could solve it. In [2], a Cruncher is programmed with the repeated squaring time lock puzzles of Rivest, Shamir and Wagner [13], and execution speeds are compared on various machines ranging from PC to Cruncher. They conclude, unsurprisingly, that some machines are much faster than others at running time lock puzzles.

Another approach to time locks is to use trusted agents. These are tamper proof computers that publish previously secret values periodically. These agents can synchronize their internal clocks by a cryptographic transaction once every few days. Besides revealing secret values periodically, these agents also respond to requests of the form, "Here are values for M and t. Please return $E(K, M)$, the encryption of message M under the secret value K which you will reveal at future time t." To use a time lock agent to distribute a patch, the vendor would make such a request to each time lock agent with M = the key for the patch. Then the vendor would send the message $(\texttt{agent_id}, t, E(K, M))$ to each site served by that agent. At time t, the site would get K from its time lock agent, use this to decipher

$E(K, M)$, then use M to decipher the patch, and finally install the patch.

Here is another variation of synchronized patch installation similar to the scheme discussed in [16]. Suppose that the patch fixes a major security hole and must be installed as soon as possible. The enciphered patch is sent to all sites and each site is told to respond when it is ready to receive the key and install the patch. As soon as a certain percentage (e.g., 99%) of the sites reply that they are ready, then the key is broadcast to all sites. However, if several (e.g., two or more) sites are compromised by rogues using this security hole, then the key is broadcast immediately and without waiting for the required percentage of the sites to reply that they are ready for it. Good authentication of messages in this protocol is essential to prevent forged "I am ready" messages. Weights could be assigned to various sites if some were thought more important than others. Then the key would be broadcast when 99% of the weighted sites were ready to install the patch.

2.4 Automatic Patch Application

Part of the OS automatically installs properly authenticated patches that it receives from the vendor over the network. The patch message authentication would have to be of the highest quality. The part of the OS that installs patches would replace some of the OS binary files. If necessary, it would then reboot the system. One problem is that different systems are configured differently, and one might have to consider this when installing certain patches and either not apply them or apply them differently on different systems. The user might not even know that his OS had been patched unless he received mail about it or he monitored the last modification time of the OS binary files. Special arrangement would have to be made to patch machines not connected to Internet.

Some users would worry about having an OS feature that allows arbitrary modification of their OS upon receipt of a special message from another computer. Many users might not care. Someone (the manager or the automatic patch applicator) should save a copy of the old unpatched OS binary file in case the patch breaks something and the new OS does not work. However, this copy would need to be saved locally — the patched version may not run so as to allow the remote patcher to revert the code base. This local copy might then enable code comparison, thus reintroducing the problem we are attempting to solve.

This is the only patch application technique that can help sites whose managers are inconsistent about installing patches, or where issues of scale are significant. System administrators are often overloaded with more im-

portant work, or ignorant of security issues, or both. At many sites, there are too few administrators to manage changes on all the machines present — especially at the rate some vendors issue patches. If the installation of security patches requires manual intervention, many system administrators often do not care about the problem until they receiveget a personal demonstration (friendly or hostile) of how seriously the security flaw can affect the security of their system. It would therefore be useful to have a tool that will automatically install security patches in a system without any system administrator intervention. Patches must of course only be installed if they have been authorized by some highly trustworthy entity, and if automatic tests before the patch installation have shown that the patch is unlikely to cause any troubles. After the patch has been performed, a number of automated tests of the fixed functionality should be performed and the patch should be undone automatically if these tests fail.

The problem with automatic patching in general is that the actual system environment found on a particular machine might vary considerably among the sites and from what the operating system manufacturer had in mind. System administrators may reorganize files and invent path structures as they feel appropriate. As few systems have good automatic software install mechanisms, the installation scripts of many software products apply in an uncontrolled way their own private patches to the system and its configuration files. Thus, it is difficult to identify a "standard" installation.

As with section 2.3, synchronized patch installation, automatic patch application does not work for machines that are turned off or disconnected from the network. The solution to this problem is the same as before: Apply the patch when the machine is rebooted or network connectivity is reestablished.

2.5 Watermarks

The idea here is to digitally sign or place a "watermark" in an OS binary file so that if a patch is reverse-engineered and someone uses the flaw it fixes to break into another site, then there will be evidence left at the compromised site pointing to the perpetrator. The patch is "signed" but not enciphered or disguised. Whether this can be done may depend on the type of flaw being fixed. This does not prevent decompilation but may be used to track which other user had decided to use this against us. These marks may not need to be highly specific for each site if they are are structured to work with other indicators such as those described by Krsul and Spafford in [9].

3 Hardware solutions

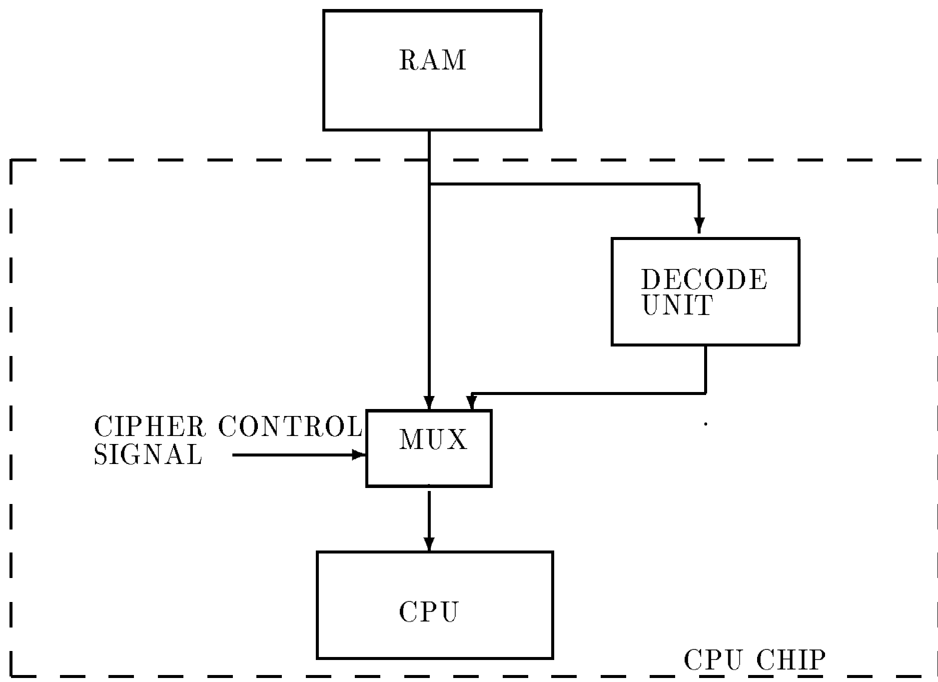
The following solutions require all user computers to have special hardware or firmware not found on conventional machines. Specifically, some or all of the instructions of the OS would be enciphered—not simply encoded—and the special hardware or microcode would decipher some or all instructions either when they are fetched from main memory or when they are loaded from disk. In the latter case, main memory would have to be protected from the users view. For example, the user could not get a core dump. By having some the code enciphered, comparisons and analysis of changes becomes much more difficult or impossible within any limited time period.

3.1 Enciphered Operating System

All OS binary files are enciphered by the vendor using a block cipher. Users receive only the enciphered binary files. To run the OS, either (a) the enciphered OS is loaded into main memory and the microcode or hardware deciphers each instruction as it is fetched or (b) the entire OS is deciphered when it is loaded into main memory and user access to it is prohibited. The patch is enciphered with the same key as the OS so that it may replace the proper OS binary files. Enciphering makes the patch unintelligible so that its installation need not be synchronized. The cipher must be simple so that performance will not be degraded. The block size must be large enough (e.g., $\geq$ 128 bits) to prevent cryptanalysis with a logic analyzer. The key might be the same for every machine or each machine might have its own key. The latter choice complicates patch distribution but provides excellent copy protection for the OS.

The diagram illustrates how the CPU fetches instructions either directly from memory or from memory through decoding hardware. A multiplexor chooses the source of the instruction.

The operating system could decrypt executables immediately prior to loading the binary image into main memory. Access control mechanisms would be necessary, which prevent even system administrators from accessing the plain text binary (for example, under Unix, `/dev/mem`, the "panic core" and similar facilities need access restrictions to memory locations that map decrypted executable pages). As the initial vector of the old and new software module version would be different, the attacker has no means to compare the old and new version, except by observing file length changes and execution timing and behavior differences. System call traces should be disabled for encrypted binaries.



INSTRUCTION FETCHES IN A CRYPTO-CHIP

This approach is not feasible for operating systems where the user has full access to the kernel source code and its compiling environment. By disassembling, the decryption keys of kernels that are only available as binaries can be identified also relatively easily. Therefore, decrypting binaries when they are loaded makes the attack a little bit more difficult, but not considerably. This concept provides no protection against retrieving the plain text software from the system bus.

3.2 Certain Modules Enciphered

A small number of OS instructions, such as a security module or part of a patch that would reveal a security hole, are enciphered. To execute programs efficiently, the enciphered instructions are placed in one segment and a segment flag tells whether its instructions are enciphered or not. Seeing this flag, the instruction decoder would decipher instructions from this segment before executing them. Since only rarely would instructions have to be deciphered, a more secure (and probably slower) cipher could be used than if all instructions were enciphered as in section 3.1.

4 Methods of Generation of Pseudochanges

In this section and the next, we illustrate some methods of making random variations in the code generated by a compiler. These are used in sections 2.1 and 2.2

When compiling a program, we use a flow graph [11] as an intermediate representation. A flow graph is a directed graph in which each node represents a basic block through which program control flows linearly and can be constructed using one of the standard techniques [1]. We perform global data flow analysis on this flow graph using Kildall's [7] scheme that associates with each statement in the program a pool of data that is being propagated through the program. This data pool is a set of expressions that are partitioned into equivalence classes such that each member in a particular equivalence class has the same value. Here is an example:

Label	Code	Data Pool (Equivalence Classes)
1	a := b	
2	d := f	{ (a,b) }
3	c := a+d	{ (a,b),(d,f) }
4	p := q+r	{ (a,b),(d,f),(c,a+d,a+f,b+d,b+f) }

Using the information in the equivalence classes we can generate equivalence-preserving changes. For example, we can replace the statement labeled

3 with any one of the following statements:

c := b+d
c := a+f
c := b+f

Each of the above statements may be used as a false change. With every false change statement that is selected in this way, we associate a label that serves as a mapping between the original program statement and the corresponding change statement. When we finish compiling the original program, we also compile the change statements and resolve all external references so that we can create the editing directives for installing the changes. We then save the binary changes along with corresponding editing directives to be used by the change applicator in our false change database. A technique for compiling this kind of changes and building up a database is described in Krauser [8], while Sayward [14] discusses the practical issues about program equivalence.

One complication with this way of false change generation occurs when a real change alters some of the original equivalence classes. If this happens, we need to distribute additional changes to correct the situation where some already distributed false changes are no longer harmless. We also need to modify the false change database.

If a program is not to be globally optimized during compilation, we can generate a significantly larger number of false changes using techniques like constant folding, copy propagation, dead code elimination, hoisting and sinking and other optimizing methods. On a related note, we can view an optimized program as simply the unoptimized program with a sequence of false changes applied to it.

5 Methods of Customization and Obfuscation of Binary Files

In this model the vendor carries out certain code rearrangement and/or modifications so that the resulting binary executable looks quite different from the unpatched version, while remaining functionally equivalent except for the patch. Here are some of the ways in which these rearrangements or modifications may be performed:

5.1 Intra-block code rearrangement

There is normally more than one way in which we can order the independent computations inside a basic block so that the resulting object code has the optimum cost in terms of instruction counts and load/stores. Such orderings are normally obtained from topological sorts of the dependence graph for a block. Aho, Sethi and Ullman [1] present an algorithm to generate optimal orderings for evaluating the nodes of a DAG representing the basic block. When applying a patch, we can reorder the computations in some of the basic blocks so that the affected blocks are still optimal, but look very different from the original blocks.

5.2 Change of control flow

We can alter the thread of execution in a program without changing its functional behavior by altering the order of execution of some of the independent basic blocks, thus altering the look of the binary executable. In the global data flow analysis phase during compilation of a program, we can generate a dependence graph between basic blocks. Any ordering of the basic block execution sequence produced by the topological sort of the dependence graph will be functionally correct.

During patch application, we can opt for an alternative execution sequence (as produced by a topological sort) for some of the basic blocks through jumps, thereby altering the binary executable. One must develop an algorithm to analyze the effect of the modified execution sequence on the register contents and to change the executable code accordingly.

5.3 Register and variable renaming

We can rename all the data registers used in the program. Interchanging some variable addresses consistently will also change the appearance of the program. One way of doing this was described in section 4. Another way is to have the GRNG select a random permutation of the variables and then rename the registers or variables according to this permutation.

Usually security patches change only a few lines of code. Sometimes only the type of a variable is corrected, one line of code is added or removed, or a branch condition is slightly modified. Because the same compiler and the same compile options are usually used to create both the old and new executable binary, we will observe only a few bytes of changed machine code. The code produced by compilers allows easy recognition of subroutine boundaries. Therefore, even if part of the machine code has been relocated

and many absolute addresses in the code have been changed, simple length comparisons and searches for the longest common subsequence will quickly identify those subroutines that have been modified. This allows the attacker to concentrate her reverse engineering efforts onto a few subroutines, which can save a lot of time.

We suggest therefore the development of the following mechanism. Take the code generation module of an existing compiler and add algorithms that allow many variations in the machine code produced. The code generator and optimizer of a compiler often make an arbitrary selection among many different machine instruction sequences that all fulfill the same purpose and that are comparable in memory and runtime efficiency. If these alternative machine sequences can be identified by the code generator, the selection of the machine code sequence actually used can be determined by a random number generator (GRNG). This way, by providing a new seed value for the GRNG as a compiler option, we can cause the compiler to generate a new executable binary, which shows in most bytes significant differences from any executable generated previously from the same or any similar source code.

The following mechanisms can (among others) be used to vary the output of machine code:

- Memory locations of variables can be permuted.
- Sequential instructions can be permuted, as long as this will not alter the program semantics. The optimizer keeps a great deal of data about how instructions depend on each other, therefore this should not be difficult to figure out.
- The order of procedures in the final code can be permuted.
- Code segments that are not marked as being in some time-critical inner loop can be generated using suboptimal but semantically identical machine sequences.
- The memory layout of code can also be reorganized by inserting many jump commands.
- Simple boolean expressions can be replaced by more complicated equivalent expressions. If the attacker tries to develop automatic software that is supposed to reverse this artificial complication, she might quickly face a number of NP-complete problems.

The compiler should support a "critical" directive to signal especially time-critical parts of the source code to exclude them from suboptimal modifications. For the rest of the software, it is perfectly acceptable if the pseudo-random variations in the code generation process cause the code produced to take some more time and memory than with the normal optimization techniques.

If the GRNG seed value is changed for every distributed software version, the attacker will find that reverse engineering only the differences between the old and new versions is at least as difficult as reverse engineering the old software version alone and searching in it for security problems. This way, the goal of secure patch distribution will have been accomplished nicely for binary files.

6 Hardware-supported decryption: cryptoprocessors

With special hardware capable of decoding an encrypted instruction before feeding it to the CPU, we may be able to apply an encrypted patch directly to the binary executable. This would prevent a user from seeing the decrypted version of the patch.

A patch will be encrypted and be applied to the binary executable in the encrypted form. CPU control logic recognizes an encrypted instruction by a special marker on the segment. In the instruction decoding phase of the execution cycle for an encrypted instruction, the routine instruction decoding will be preceded by a decrypting step in which the encrypted instruction will be decoded by a hardwared decoding unit with an embedded decryption key.

To avoid having a longer clock cycle time because of the decrypting phase, we may prefetch some of the encrypted instructions and pipeline them through the decryption unit. To keep the decryption pipelining scheme simple, we may leave the branch instructions in the patch unencrypted in the first place.

Apart from the cost of the additional hardware, this scheme requires some central authority to decide what the encrypting and decrypting keys will be.

Why would users buy a cryptoprocessor — a machine that executes encrypted programs? One marketing advantage is that software would be cheaper for a cryptoprocessor because the vendor knows that it can be used on only one machine. Copy protection is enforced.

The ideal solution would somehow have to avoid having anyone outside the software development team get access to the plain text version of the software, both the old unpatched and the new patched version. That would certainly provide the highest level of security and would at the same time allow effective software piracy prevention. Mechanisms that completely prevent (even hardware) access to the executed software include:

- Secure main board packages as implemented in the ABYSS system [18]. The CPU, the RAM, and some peripheral devices are all enclosed in a tamper-proof package. Software is stored in encrypted form on a hard disk outside the security shield or loaded in encrypted form over a network. The (machine specific) decryption keys are stored in a battery buffered RAM inside the secure package. The software is decrypted when it is loaded from external storage into the RAM. The secure package prevents hardware observation of the decrypted software in the system RAM or on the system bus lines. The operating system kernel is also loaded encrypted into the machine and can therefore not be modified to release the protected software.

- Cryptoprocessors perform the decryption between the memory interface of the CPU chip and the on-chip cache. The security package is limited to the CPU package, which simplifies manufacture and servicing, and avoids memory capacity limitations. Cryptoprocessors have first been described in [3] and existing implementations include the DS5002FP microcontroller and the iPower security processor. A cryptoprocessor concept suitable for operation in a modern multitasking workstation, in which it is not even necessary to trust the operating system, is the TrustNo1 processor concept described in [10].

Although cryptoprocessors provide in our opinion the basis for the most secure patch distribution concepts, they are at the moment more of academic interest, because they are currently available on the civilian market only for microcontroller applications and there exists today no cryptoprocessor for personal computer applications. Therefore, the cryptoprocessor concept should be considered as an ideal solution and should be documented as a reference for systematic comparison with other patch distribution concepts, but considering the lack of existing hardware, these concepts are probably not what we should recommend in the near future. We could of course consider developing such a chip based on an existing microprocessor design.

7 The costs

It is not easy to modify a compiler to make it use a GRNG to determine code arrangement, register assignments, etc. Thus, the methods described in section 2.1, Customization, and section 2.2, Obfuscation, have a high cost in development. Customization has the additional cost of compiling the program once for each customer, each compilation using a different seed for the GRNG. If there are tens of thousands of customers and hundreds of thousands of lines of code to be compiled, this cost will preclude the use of customization. Customization could be made feasible by customizing the OS only for a special class of customers who pay for this service. The vendor would compile the patch once for each special customer (each time with a unique GRNG seed) and once more for all regular customers together (using one more GRNG seed). In contrast, obfuscation requires that the OS be compiled only once per release. One vendor (HP) maintains a database of customer options that might serve as a model for the record keeping needed for customization.

A major cost of synchronized patch installation, as described in section 2.3, is the creation of the patch distribution hierarchy. Of course, patches are already distributed now. Perhaps a slight modification of the present system would suffice. If cryptography is to be used, then an appropriate cryptosystem must be chosen and implemented; the political and key management issues likely make this solution unworkable at present. Likewise, time locks would add to the cost if they were used.

One cost of automatic patch application, as described in section 2.4, is the development of the OS module that applies patches and the authentication system it uses. Another cost is the creation of a patch application hierarchy similar to the patch distribution hierarchy above. The installation module must know which features of the OS were selected when the OS was created so that it will not try to patch a non-existent module. It must also know which version of the OS is currently running. Customers should be able to undo a patch that they do not want. The people we have asked about this approach overwhelmingly said they did not want automatic patch application on their systems.

The cost of the hardware solutions are the special hardware, firmware or software to decipher instructions and/or protect main memory from direct user access. There are also the costs of the cipher, of key management, and of enciphering many copies of the OS. The latter cost may be as prohibitive as that of compiling many copies of the OS as in section 2.1. What if a customer bought many systems? Would they have different keys or the

same key? An additional cost of the method described in section 3.2 is the redesign of the OS to put all security functions in one segment.

8 Present practice

What do vendors currently do about patches? We asked some, and here is a summary of their responses. (All of the vendors who responded spoke with us only on condition of anonymity.)

One vendor does not issue patches. The only software changes are new releases of the OS. (By observation, this is true for several vendors.)

Another vendor simply issues the patch and forgets about it. If the patch fixes a security flaw, the patch starts with the message, "This patch fixes a security flaw. Install it now or else the consequences are your problem." This vendor guesses that about half of its customers actually install patches.

A third vendor built a prototype of an automatic patch installation system similar to that described in section 2.4. It was never put into use because a survey of their customers found that they would have nothing to do with it. This vendor uses the following system to distribute patches: A service agent calls customers who pay for patch service and tells them what patches are available. Over the telephone, the customers select the patches they want, and the service agent sends these patches to them by FedEx. In addition, all patches are posted on a web site from which any customer can download whatever she wants. As with the second vendor, security fixes carry a message, "Install this soon or else it is your problem."

9 Our recommendations

Based on our study to date, we recommend automatic patch application provided users can be made to accept it. It is effective and its costs are moderate. If users will not accept it, then our second choice is a combination of the techniques presented in sections 2.2 and 2.3. The cost should be only slightly more than the automated application method and it would be nearly as effective. The enciphered OS approach may become feasible some day if vendors produce cryptoprocessors to prohibit program copying.

10 Future work

Many of the issues examined in this paper raise more questions than we answer here.

How much of this paper applies to security tools, too? Consider the issues raised in section 2.1, for example. Could we customize a password checker to make it work only on one machine? If an OS were customized, would an audit tool have to be customized in a compatible way? Some questions which arise in that section are where in a compiler to use the random numbers (intermediate code or final code generation), what are the best ways to make random choices, and how this may affect program efficiency? Eventually, we might produce a compiler with this sort of GRNG controlling its code generation. We discuss some of this in other sections of the paper, but the issues are not resolved..

These questions arise in section 2.2: Can we show it is NP-hard to find a security fix in this collection of changes? Or can we think of any disassembly tools that would facilitate discovery of the real security fix? How good are disassemblers? Are zero-knowledge proof techniques relevant here? Can one use program mutation techniques [14] to generate the false changes? Many mutants are equivalent and may be used to generate pseudochanges. Are 100 changes enough?

The automatic patch installer of section 2.4 is a highly system-dependent mechanism. Some vendors (e.g. SunSoft) offer already comfortable and semiautomatic patch installation systems. We could develop a completely new state of the art automatic patch distribution system for one specific environment, and document its design concepts, the practical experiences, and the unresolved problems in some papers. Alternatively, we could try to improve existing semiautomatic patch systems with additional functionalities towards fully automated operation.

Here are some questions concerning Section 3. What ciphers should be used? When the program is swapped out, should its data variables be enciphered? How do we recover or repair system "crashes" or component failures if we cannot recover the key? Can we combine this mechanism with other cryptographic needs on the system.

We hope to be able to answer some of these questions with our future research.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1988.

[2] Lyle T. Bertz, Ganesh Krishnan, Markus G. Kuhn, E. H. Spafford, and S. S. Wagstaff Jr. Remarks on time locks. In preparation.

[3] Robert M. Best. Preventing software piracy with crypto-microprocessors. In *Proc. IEEE Spring COMPCON 80 San Francisco, California*, pages 466–469, February 25–28, 1980.

[4] Chris Caldwell. The Dubner PC Cruncher–a microcomputer coprocessor card for doing integer arithmetic. *J. Recreational Mathematics*, 25(1), 1993. This hardware is available from H & R Dubner, 449 Beverly Road, Ridgewood, New Jersey 07450.

[5] Donald E. Eastlake, Stephen D. Crocker, and Jeffrey I. Schiller. *RFC–1750 Randomness Recommendations for Security.* Network Working Group, December 1994.

[6] Daniel Farmer and Eugene H. Spafford. The COPS security checker system. In *Proceedings of the Summer Conference*, Berkeley, CA, June, 1990. Usenix Association.

[7] Gary Kildall. A unified approach to global program optimization. *Conference Record of ACM Symposium on Programming Languages*, pages 194–205, 1973.

[8] Edward Krauser. *Compiler-integrated Software Testing.* Ph. D. thesis, Purdue University, 1991.

[9] Ivan Krsul and Eugene H. Spafford. Authorship analysis: Identifying the author of a program. In *18th National Information Security Conference*, volume 2, pages 514–524, October, 1995.

[10] M. Kuhn. *Sicherheitsanalyse eines Mikroprozessors mit Busverschlüsselung.* Diploma thesis, Lehrstuhl für Rechnerstrukturen, Universität Erlangen-Nürnberg, Erlangen, July, 1996.

[11] Karl Joseph Ottenstein. *Data-flow graphs as an Intermediate Program Form.* Ph. D. thesis, Purdue University, 1978.

[12] Tim Polk. Automated tools for testing computer system vulnerability. Technical Report NIST SP 800–6, National Institute of Standards and Technology, 1993.

[13] Ronald L. Rivest, Adi Shamir, and David A. Wagner. Time-lock puzzles and timed-release crypto. Prepreint, 9 pages.

[14] F. Sayward and D. Baldwin. Heuristics for determining equivalence of program mutations. Research Report 161, Georgia Institute of Technology, April, 1979.

[15] Eugene H. Spafford. Viruses. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 2, pages 1433–1441. Wiley-Interscience, 1994.

[16] Eugene H. Spafford. The pros and cons of disclosure. In *Conference on Systems Administration and Network Security*. USENIX, May, 1995. Invited address not in proceedings.

[17] Cliff Stoll. Telling the goodguys: Disseminating information on security holes. In *Proceedings of the Fourth Aerospace Computer Security Conference*, pages 216–218, Washington, DC, 1988. IEEE Computer Society.

[18] Comerford White. ABYSS: A trusted architecture for software protection. In *Proc. 1987 IEEE Symposium on Security and Privacy, Oakland, California*, pages 38–51. IEEE Computer Society Press, April 27–29, 1987.