# 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

## 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 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, 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

[*] Current address: Alamadanga Kushtia, Bangladesh.

one with which most people are familiar, and presents sufficient complexity and risk to warrant concern. In addition, we discuss how some of our solutions may 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 [8] and [5], 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 consider cannot be solved in its entirety. Therefore, we must also consider ways to reduce the associated risk.

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

- We can “customize” each patch or tool so that each one differs from machine to machine.
- We can introduce “noise” to mask changes.
- We can synchronize patch distribution and application so that all users receive and install the patch at the same time.
- We can use automated patching: Part of the operating system patches itself when it receives an authenticated command over the network from the vendor.
- Use cryptographic methods to obscure patches.

In what follows, we classify solutions 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.

## 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. [4]). 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. The patch is compiled using the same random numbers as the module it fixes.

As different sites have slightly different OSs, they might have different forms of a flaw and require different patches. 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 form of the bug. For example, if the flaw were a buffer overflow, then different versions of the OS might have different offsets from the buffer to a variable or stack return address to overwrite. 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. However, now the patch is compiled using new random numbers. The GRNG could also be used to introduce unnecessarily complex 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. 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 from the GRNG. The malicious user who compares the old and new binary files will

[1] A pseudo-random number 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

find thousands (or more) of differences and thus have great difficulty discovering the security flaw. This way, reverse engineering the patch becomes almost as difficult as reverse engineering the entire original vulnerable version.

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. 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.

## 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 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 signed 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 plain text) instructions to install it at noon Universal Time (GMT) on a certain day, at which time the key to the cipher will be revealed. Cliff Stoll first described this scenario [12] 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*. Today the WWW might be a more appropriate medium due to its worldwide coverage. One might use some form of time locks (first suggested in [11] and independently developed in [9]) 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 personal computer and another a Cruncher [3], which can solve "time lock puzzles" requiring arithmetic with large integers hundreds of times faster than a PC could solve it. In addition, the "time lock puzzle" must be distributed synchronously, which is the same as the original problem of distributing the patch keys.

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 (`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.

A variation of these trusted agents would be for a standard time service such as WWV, DCF77 or NTP to broadcast a signed time stamp periodically, say, once per hour. The trusted agent is a smart card which contains the public key of the time service and the patch key. It reveals the patch key as soon as it receives the signed time stamp for a certain hour (or a later hour). The smart card has a session key known only to it and the vendor, and this key is used to load the patch key into the smart card, along with the release hour.

[2] We should note that this loosely corresponds to the current logical organization of FIRST, the Forum of Incident Response and Security Teams.

## 2.4 Automatic Patch Application

Part of the OS automatically installs properly authenticated patches that it receives from the vendor over the network. Of course, the patch message authentication would have to be of the highest quality and the user would have to trust the vendor. 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 we must allow the remote patcher to revert the old OS.

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 important work, or ignorant of security issues, or both. 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.

# 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. Software protection is not sufficient. The hardware must be physically protected, for example, from a malicious user attaching a logic analyzer to a bus.

## 3.1 Enciphered Operating System

All OS binary files are enciphered by the vendor. A block cipher would be best in this application, to provide random access to the enciphered instructions. 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 denied by locating both RAM and processor in a tamper resistant module. 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 as well as for application programs that use the same mechanism. The CPU would fetch instructions either directly from memory or from memory through decoding hardware. A multiplexor chooses the source of the instruction.

## 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 Customization and Obfuscation of Binary Files

In this model, which we summarized in sections 2.1 and 2.2, 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.

## 4.1 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. Also, 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, especially since the instruction level optimization will often select very different instructions after some reordering.

## 4.2 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.

## 4.3 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. 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 (CRNG). This way, by providing a new seed value for the CRNG 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 CRNG 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.

# 5 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 hardwired 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[13]. 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 [2] and existing implementations include the DS5002FP microcontroller and the iPower security processor. Another important reference for cryptoprocessors is [6]. 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 [7].

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.

## 6 The costs

It is not easy to modify a compiler to make it use a CPRNG 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 tool development. Customization has the additional cost of compiling the program once for each customer, each compilation using a different seed for the CPRNG. 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 CPRNG seed) and once more for all regular customers together (using one more CPRNG 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, especially for a global customer base. 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 again the creation of a patch distribution hierarchy. 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, either because of security risk or because the high frequency of modification hinders isolation of fault causes.

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. This includes the cost of tamperproof packaging. 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.

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

Another 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 express mail. 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.'

## 8 Our recommendations

Based on our study to date, we recommend automatic patch application provided users can be convinced 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.

## 9 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 CRNG 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 [10] 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. In particular, we would like to evaluate our approaches using a well-documented security threat.

## 10 Acknowledgement

Portions of this work were supported by contract F30602-96-1-0334 from Rome Laboratory (USAF), by sponsors of the COAST Laboratory.

## References

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

[2] 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.

[3] 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.

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

[5] Daniel Farmer and Eugene H. Spafford. The COPS security checker system. In *Proceedings of the Summer 1990 Usenix Conference*, pages 165-170, Berkeley, CA, June, 1990. Usenix Association.

[6] S. T. Kent. *Protecting Externally Supplied Software in Small Computers*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, March, 1981. MIT Laboratory for Computer Science, MIT/LCS/TR-255.

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

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

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

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

[11] 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.

[12] 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.

[13] 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.