Katana: Towards Patching as a Runtime part of the Compiler-Linker-Loader Toolchain

Sergey Bratus, James Oakley, Ashwin Ramaswamy, Sean W. Smith[1]
Computer Science Dept.
Dartmouth College
Hanover, New Hampshire

Michael E. Locasto[2]
Computer Science Dept.
George Mason University
Arlington, Virginia

ABSTRACT

Despite advances in modularity, security, and reliability of software, offline patching remains the predominant form of updating or protecting commodity software. Unfortunately, the mechanics of *hot patching* (the process of upgrading a program while it executes) remain understudied, even though such a capability offers practical benefits for both consumer and mission-critical systems. A reliable hot patching procedure would serve particularly well by reducing the downtime necessary for critical functionality or security upgrades. Yet, hot patching also carries the risk – real or perceived – of leaving the system in an inconsistent state, which leads many owners to forgo its benefits as too risky; for systems where availability is critical, this decision may result in leaving systems unpatched and hence vulnerable. In this paper, we present a novel method for hot patching ELF binaries that supports *(a)* synchronized global data and code updates and *(b)* reasoning about the results of applying the hot patch. We develop a format, which we call a *Patch Object*, for encoding patches as a special type of ELF relocatable object file. We then build a tool, *Katana*, that automatically creates these patch objects as a by-product of the standard source build process. Katana also allows an end-user to apply the Patch Objects to a running process. In essence, our method can be viewed as an extension of the Application Binary Interface (*ABI*), and we argue for its inclusion in future ABI standards.

Keywords: hotpatch, patch, ELF, DWARF, reliability, toolchain

---

1 Supported in part by the National Science Foundation, under grant CNS-0524695. The views and conclusions do not necessarily represent those of the sponsors.

2 Supported in part by grant 2006-CS-001-000001 from the U.S. Department of Homeland Security under the auspices of the I3P research program. The I3P is managed by Dartmouth College. The opinions expressed in this paper should not be taken as the view of the authors' institutions, the DHS, or the I3P.

I INTRODUCTION

It is somewhat ironic that users and organizatations hesitate to apply patches — whose stated purpose is to support availability or reliability — precisely *because* the process of doing so can lead to downtime (both from the patching process itself as well as unanticipated issues with the patch). Periodic reboots in desktop systems — irrespective of the vendor — are at best annoying. Reboots in enterprise environments (*e.g.*, trading, e-commerce, core network systems), even for a few minutes, imply large revenue loss — or require an extensive backup and failover infrastructure with rolling updates to mitigate such loss.

We question whether this *de facto* acceptance of significant downtime and redundant infrastructure should not be abandoned in favor of a reliable hot patching process.

Software, the product of an inherently human process, remains a flawed and incomplete artifact. This reality leads to the uncomfortable inevitability of future fixes, upgrades, and enhancements. Given the way such fixes are currently applied (*i.e.*, patch and reboot), developers accept downtime as a foregone conclusion even as the software is released — and deployers who resist downtime resist the patches.

While patches themselves are a necessity, we believe that the process of *applying* them remains rather crude. First, the target process is terminated; the new binary and corresponding libraries (if any) are then written over the older versions; the system is restarted if necessary; and finally the upgraded application begins execution. Besides the appreciable loss in uptime, all context held by the application is also lost, unless the application had saved its state to persistent storage (Candea and Fox, 2003, Brown and Patterson, 2002) and later restored it (which is expensive to design for, implement, and execute). In the case of mission-critical services, even after a major flaw is unveiled and a patch subsequently created, administrators must choose between security (applying a patch) and availability. This conundrum serves as our motivation for *hot patching*, without restarting the program and losing state and time. We focus on systems, such as those found in the cyber infrastructure for the power grid, which require high availability and which store significant state (that would be lost on a restart).

Challenges of Patching.

Requiring and encouraging the adoption of the latest security patches is a matter of common wisdom and prudent policy. It appears, however, that this wisdom is routinely ignored in practice. This disconnect suggests that we should look for the reasons underlying users' hesitancy to apply patches, as these reasons might be due to fundamental technical challenges that are not yet recognized as such. We believe that the current mechanics of applying patches prove to be just such a stumbling block, and we contend that the underlying challenges need to and can be addressed in a fundamental manner *by extending the core elements of the ABI and the executable file format*.

Mission-critical systems seem hardest to patch. They can ill afford downtime, and the owner may be reluctant to patch due to the real or perceived risk of the patch breaking essential functionality. For example, patching a component of a distributed system might lead to a loss or corruption of state for the entire system. An administrator might also suspect that the patch is incompatible with some legacy parts of the system. Even so, the patch may target a latent vulnerability in a software feature that is not now in active use, but also cannot be easily made unreachable via configuration or module unloading. The administrator is forced to accept a particularly thorny choice: inaction holds as much risk as a proactive "responsible" approach. Since the risks of patching must be weighed

against those of staying unpatched, we seek to ***shift the balance of this decision toward hot patching by making it not only possible, but also less risky in a broad range of circumstances***. We contend that this can only be done through good engineering and making patching a part of the standard toolchain.

Our key observation is that current binary patches, whether "hot" or static, are almost entirely opaque and do not support any form of reasoning about the impact of the patch (short of reverse engineering both the patch and the targeted binary). In particular, it is hard for the software owner to find out whether and how a patch would affect any particular subsystem in any other way than applying the patch on a test system and trying it out, somehow finding a way to faithfully replicate the conditions of the production environment.

Given these circumstances, our tool Katana and our Patch Object format not only seek to make possible the mechanics of hot patching, but also enable administrators to reduce the risk of applying a particular fix by providing them with enough information to support examination of the patch structure, to reason[1] about its interaction with the rest of the system, and to understand the tradeoffs involved in applying it.

Patching In The Toolchain.

Hot patching should not be thought of as a bizarre operation, done crudely and infrequently. We argue that it is one of the fundamental transformations in the life cycle of any program, along with compilation, linking, dynamic linking, and (in unfortunate cases) dumping core. We note that each of these fundamental operations has its own type of ELF object devoted to it (relocatable objects, executables, dynamic libraries, core dumps respectively) and a corresponding tool in the toolchain for performing each transformation or working with its output. *We contend that patching is very much like linking or dynamic linking*. Like those operations, it combines (or replaces) parts of programs and must generate, modify, and apply relocation information. The section types defined for the ELF format contain nearly all of the information necessary to describe a patch. Once we take the position that patching is like linking, it follows that a patch needs to store the same type of symbol and relocation information as does any relocatable ELF object.

What the base ELF specification lacks is a way to describe types. Symbol information gives us only a location and a length, but there is no way to describe the internal layout of a piece of data. The DWARF format,[3] already heavily used with ELF for debugging and exception-handling purposes, provides exactly what is needed here as it provides a means to recursively describe types and variables, as well as a set of instructions originally designed for restoring register states and examining the call stack but rich in possible applications.

Through the use of formats already employed in the binary tool chain, we hope to promote easy examination of patches, interoperability, and to show that patching fits comfortably into the rest of the toolchain. We therefore propose that the standard software life cycle now be as shown in Figure 1. Before hot patching, only the Development and Runtime stages of the figure were generally accepted.

<figure1.tiff GOES HERE>

Figure 1: Revised software life cycle. Before hot patching, only the Development and Runtime portions existed

Why Not Just Employ Redundancy?

Redundant infrastructure, containing replicas of nodes and service paths, often helps an organization bridge the service disruption stemming from patches. We believe, however, that redundancy isn't always the best approach for ensuring availability during an upgrade or security-critical patching process. Rather than an established best practice, we invite the reader to see redundancy as an extreme measure that needlessly duplicates hardware, networking, and software of the original system. We suggest that redundancy is:

1. **expensive -** In medium-sized enterprises, the cost of a single server, gateway, or switch is high enough to outweigh the benefits of redundancy.
2. **wasteful -** Redundant systems are typically passive bystanders, lying in wait for an active machine to initiate a failover.
3. **dependent upon complicated logic -** Transferring application state (even across multiple homogenous systems) is non-trivial, especially when the state transfer occurs within hardware (such as for call trunks).
4. **specialized -** The process of building system redundancy is not easily generalizable across heterogenous systems and requires full knowledge of the underlying protocol and application state in order to provide faithful failover and failback.

It should be noted that the last two items apply even for virtualized redundant systems, which often do not have the traditional overhead of redundant hardware. We do not claim that redundancy does not have its place, but redundancy does not provide the easy, ubiquitous solution to high-availability stateful applications that we hope to provide through patching.

2 KATANA DESIGN

In our prototype, a patch is may be generated as follows. The source directory corresponding to the target to be patched is replicated, and the source code is patched to the version desired. The modified source tree is then built and compared with the original source tree at the object (.o) level. Those object files that have changed between the modified and original source trees are added to the list of objects that must be examined for type and code transformations. (Future work will include use of the *inotify* mechanism to avoid potentially expensive recursive directory comparison and provide more precise notification of changed files.)

<figure2.tiff GOES HERE>

Figure 2: *An Example Code Base. From the top: each source file creates a corresponding object file; multiple object files are combined into intermediate compilation units (CU); and multiple CUs are merged to form the executable. All shaded blocks indicate modified files.*

To dynamically update the running application, Katana needs to patch both the **code** and

the **data** within the process. It first creates a *patch object* (PO): an ELF file with sections that indicate the type of patch (code or data), the patch offsets and lengths within the process address space, patch data, function and data names, etc. The patch object may then be applied to the target at any time.

3 AUTOMATED PATCHING

In this section, we describe our data and code patching methods. We note that, compared to previous work, our PO data structures allow reasoning about the scope, extent, and impact of the patch (*e.g.,* whether it affects particular subsystems within the process).

Code Patching

This process involves several stages:

(i) *Code Identification:* Katana first needs to identify the section(s) of text that need to be modified within the running process. To do this, we consider the list of all modified object files from our tracking step and identify all functions (both static and global) within these files from their symbol table. Functions that differ between the original and modified versions of an object file are copied into the PO and marked as code.

(ii) *Symbol Resolution:* After identifying all functions that require a patch, we need to resolve outstanding symbol references within each function. Typically, symbol resolution for an application happens at both the linking stage (called *static linking* when the symbol is present within another object file or archive), and the execution stage (or *dynamic linking*, when the symbol is present within a shared library). All code relocations are identified in the ELF sections `.rel.text` and `.rela.text`, within the object files and the final executable. Each relocation entry contains, among other information, the code offset that requires relocation and the outstanding symbol that provides this fix-up.

For each relocation entry, Katana copies the corresponding symbol into the PO. The actual value of the symbol is not necessary in the PO unless the target to be patched will be stripped, as the value of the symbol can be retrieved from the running target while performing the patching. This is key in allowing executables that have already been patched to be patched again. If the symbol was dynamic (*i.e.,* present in a shared library such as `libc`), then the fixup value is the address of a corresponding entry in the procedure linkage table (*PLT*) of the executable. The PLT is essentially a jump table with entries for each symbol that needs to be resolved at runtime by the dynamic linker. When the process begins execution, the dynamic linker maps the required shared libraries into the address space of the process and updates each PLT entry.

For dynamic symbols, Katana traverses the PLT entries of the executable and compares the symbol name of each entry with the symbol name that requires relocation. Once a match is found, the symbol value can be determined. Our current prototype cannot add calls to previously unused functions in shared libraries, but support for this will be added using the "ALTPLT" technique described in *Embedded Elf Debugging : The Middle Head of Cerberus* (The ELF shell crew, 2005).

Finally, if the outstanding symbol's definition was not found within the replicated executable (either within the symbol table or within the PLT), then it was newly added by the patch; it is marked as such and added to the PO.

(iii) *Patch Application:* Applying a code patch is simple enough and has been researched in other systems (Ikebe and Kawarasaki, 2006, Ukai, 2004, Arnold and Kaashoek, 2009, Yamato and Abe, 2009). We map the new function in memory and insert a trampoline `jmp` instruction at the beginning of the old function within the process

memory image. This interposition allows the caller to execute our new function instead of the previous one at the cost of an extra jump. It is possible to avoid the overhead (from branch mis-prediction) of the `jmp` instruction by adding into the old function code that traces up the stack and modifies the caller's `call` instruction operand to point to the new address instead of the old one. Although this optimization would ensure that all subsequent calls from the same caller would execute the new patched function without stepping into the old one, it does makes the process of rolling back a patch non-trivial. A simpler method to avoid the overhead would be to relocate all calls to the function to point to its new definition (although the trampoline would still be desirable, to catch calls from function pointers or anything of similar nature). The current prototype uses only the trampoline method.

Data Patching.

Patching data within a running process is significantly harder than patching application code. The primary challenge here is to synchronize the code and the data structures it acts on.

Tracking down previously allocated data is nontrivial (one of the reasons why garbage collectors are interwoven with the language implementation). Even after identifying the allocated chunks of memory, in the absence of some kind of type specification, the *internal structure* of memory remains opaque. We also need a method for extracting only the modified data variables from the patch and a means to discover the actual modifications that were performed. Our system solves both of these problems.

We first note that any code that acts on patch-modified data is already taken care of by Katana's code patching process because we rely on `make` to build the object files that correspond to all modified sources. We resolve the previously identified problems towards patching data by leveraging DWARF debugging information within the application executable. This requires the object files to be compiled with debugging support, but we do not see this as a limitation. Since we need DWARF information only while building the PO, all debugging symbols could be stripped from the executable during application deployment, if desired (this would require storing symbol values in the PO, however).

We now recall the representation of types in the DWARF format and then detail the various steps in Katana's data patching process.

DWARF type information. The DWARF structure is laid out as a tree of DIEs (*Debugging Information Entries*) within the executable file. Each DIE has an associated tag and a set of attributes. The DIE that defines type information has the tag as one of `DW_TAG_base_type`, `DW_TAG_structure_type` or `DW_TAG_union_type`. Typedefs and other type modifiers (such as `const`, `volatile`, `pointer` etc.) are referenced by the DIE that defines the type. In case of structures or unions, each member is contained as a separate DIE within the parent DIE that identifies the struct/union. It is important to note that DWARF annotates types of *all* visibilities from the program sources - local, global and static.

Katana's data patching process contains a number of steps:

(i) *Type Discovery:* We set out to discover all newly created or modified data types – those that are primarily user-defined (such as structures and unions in C). Katana traverses the type information (as identified by the above DWARF tags) from the newly created executable, and for each encountered type, it searches for the corresponding type-name within the *replicated* executable (from before the patch). If so found, the full types (i.e. the number, type and position of all member variables contained within) are

compared to determine if they are identical. If not identical, a transformation between the old and new versions is generated, along with DWARF information identifying the type to insert into the PO as soon as a variable is found making use of the altered type. Else, if the type name itself was not found within the replicated executable, then the current type was created by the patch, and is added as such to the PO.

(ii) *Data Traversal:* The next step is to traverse all variables defined within the new application, and for each one encountered, we first determine its lexical scope. If the scope is local, then we ensure that the corresponding function (the one that defines this variable) does not have an activation frame on the program stack while applying the patch. Else, the variable has been defined as either global or static. We first check whether the replicated executable defines the same variable. If not, then this variable has been created by the patch and we need not worry about it and may leave the symbol resolution up to the compiler (as only *new* code can use this variable). Otherwise, we verify whether the variable's type is one of the modified types identified during *type discovery*. If it is, then we add the variable along with its *original* address from the replicated executable, its *new* address from the patch, and its type information to the PO. At the end of this stage, Katana would have identified all newly created or modified variables from the patch.

(iii) *Patch Application:* Applying a data patch consists of first tracking down the relevant symbols in program memory. Katana reads in the PO, and for each data variable encountered, it checks whether the variable is a pointer or not. If it is, then the current validity of the pointer is verified (by bounds-checking the pointer value to within heap boundaries). If the pointer is found to be invalid, no further action is taken. If the pointer is valid, then memory for the new type(s) is allocated, the older structure is copied into the new one taking into account the difference in structure definition, the old memory is then freed, and the pointer is modified to point to the new segment (in case of structures such as lists, trees, since we have the type specification, we can repeat this process recursively for each node on the list or tree). Else if the variable is not a pointer, then Katana modifies all its references in the program text to the updated memory location from the patch. Katana supports default values in the sense that if the variable has an initializer in the new version of the program text, that initializer will be used for any member variables within structures or array elements that did not previously exist in the executing target. Initializers from the new version are used for all `const` global variables. Eventually Katana it will support default values and programmer-written custom initializers.

Challenges.

Data patching, as described above, is not an easy task, even with DWARF type information. C, as well as many other directly compiled languages, does not have a strong type system, and is not designed to allow reflection. Structures are relatively straightforward to patch, because they contain a detailed specification. Unions and arrays, on the other hand, are generally quite opaque. If substantial changes have been made to one of the types in a union, Katana cannot do anything automatically, as there is no automatic way to determine which unioned type to act on. If an array changes size, Katana will assume that it is growing or shrinking at the end and will copy old data accordingly, but it will also issue a warning that this may not be the desired behavior. Pointers are handled, of course, but with some limitations currently. Unfortunately for our purposes, memory management is not part of the ABI. Further, there is no standard way to determine what block a given pointer is part of. Our current implementation

correctly handles only memory management with `malloc` and only pointers to the beginning of blocks. Improvement of this is a major area of future work. Multiple pointers to the same address are handled by keeping track of which addresses have been relocated. `void*` also poses a problem, as there is generally no way to determine the "real" type of the pointer. The general solution to all of these type problems is to ask the programmer for routines which perform the application-specific work. Minimizing programmer work is of foremost importance, because greater human interaction adds greater possibility for human error, thus possibly decreasing the reliability of the patch, but it is not avoidable in all situations.

Hot patching still faces a number of other challenges, including dealing with multithreaded programs and address space randomization (which slight changes to the OS loader can help us overcome). There is nothing inherent in our design which will not work with multithreading, but dealing with it through ptrace takes considerable work which we have not yet done. More importantly, deadlock avoidance work — required to ensure that we do at some point pause all threads but that we never pause a thread that another thread may be waiting on in order to reach a safe state (see below) — is nontrivial. We plan to address this in the future.

4 DISCUSSION

WHEN TO APPLY THE PATCH.

Dynamically updating a running application requires diligence and patience. One cannot update the target application without any knowledge of the program's execution state, by which we mean the program stack, processor registers, etc. Even after possessing this information, the application has to be in what we call a "safe state" for Katana to apply the patch. We characterize a program state as a *safe state* if the following two conditions hold:

- All activation frames in the program stack belong to functions that *do not* get updated during code patching. It is easy to verify this by comparing each function on the stack with the list of upgradeable functions contained within the PO.
- All activation frames in the program stack belong to functions that do not access any global/static symbols identified during *Data Traversal* and that do not define any local variables of the modified types identified during *Type Discovery*. Again, since we maintain type and variable definitions within the PO, verifying this condition is easy.

Given the patch object, Katana uses Linux's `ptrace` interface to temporarily halt the execution of the target process, query the current execution stack, and determine whether the application is currently in a safe state. If so, then Katana applies the code patch followed by the data patch. We note that it is not possible to apply the code and data patches at different times since new code likely uses the new data, and hence postponing data patching to when only the second condition is satisfied is impractical and unsafe.

Let *A* denote the current activation frame on the program stack, and the notation (*X*:*Y*) define all frames from *X* to *Y* (on a time scale, *X* precedes *Y*) on the stack; so (1:*A*) defines the current stack. [**ARE X AND Y FRAMES? AND WHAT IS "1"?]** Now, in case we determine (1:*A*) to be an unsafe state, we could repeatedly keep querying the stack until the application reaches some safe state. However, this is highly inefficient and cumbersome.

Instead, when we determine the program to be in an unsafe state, we traverse up the stack from *A*, and for each preceding frame (say *A*'), we determine if (1:*A*') is a safe state. This takes into consideration only *A*' and all other frames preceding it. If (1:*A*') is a safe state, then we insert a breakpoint on the return instruction pointer (`EIP`) pointed to by the successive frame: (*A*'+1). What this guarantees us is that when this breakpoint is hit, the state of the program stack will be (1:*A*'). Since we just determined this to be a safe state, we can reliably conduct the patching procedure. If however, no such frames preceding *A* satisfy a safe state, then it means that the application cannot be patched successfully in its current execution since there will always be a function that violates our safety condition.

Finally we note that even after inserting a breakpoint, the problem of determining *when* the breakpoint will be hit is essentially a hard one, and so in such cases, Katana can provide no time-bounding guarantees. Still, this is a cleaner and more efficient approach than just naïvely retrying the full update procedure, which is both an expensive and an incomplete solution. In cases where the timely (or even eventual) application of a patch seems unlikely, it would be possible to add support for programmers to manually specify safe places for patching to occur. For programs built around a central loop (the vast majority of long-running programs), the likelihood of safe patching can be expedited by isolating the main loop into a simple routine unlikely ever to need patching. While safety determination is implemented in our prototype, we do not yet have enough real data about the likelihood of applications' failing to reach a safe state.

Address Space Randomization.

Load-time address randomization has become a stable and popular way of raising the bar for attackers, and so we must discuss how it interacts with our patching scheme. The gist of randomization schemes is invalidating various default assumptions regarding the locations of code and data elements that might facilitate exploitation. In particular, the virtual addresses of loadable segments are displaced by random[4] offsets by the loader, which *relocates* them (using their accompanying relocation sections).

Patching relocated code with our PO files requires knowledge of the displacements introduced at loading-and-relocation time. While there is no common ABI standard for saving this information, conceptually it is no different from saving virtual addresses of other files' loaded symbols in the *Global Offset Table* (GOT). We note that the names of the constituent object files themselves are customarily included in the symbol tables and that the symbol table entry format can be easily adopted for storing virtual addresses of the relocated objects.

Thus, at the cost of small modifications to the OS loader and the dynamic linker, we can make the information on the layout of the "randomly" relocated executable and libraries available to our patching process driven by our POs.[5]

Future Work.

Katana is a work in progress. We have demonstrated code patching (including dynamically linked functions), the use of our patch object format (discussed in detail in Section 5 below), and data patching including complex structures and variable addition. The remainder of the system is still under development. In the future we must address several important engineering issues, such as the interaction of patched code with dynamically loaded libraries (including the *dlopen* mechanism) and assuring that accumulation of administered patches does not lead to unacceptable performance degradation. We must also address the broader issue of describing and detecting software

designs not amenable to runtime patching, and we must steer programmers to avoid them if possible.

5 PATCH OBJECT FORMAT

Reasons and Needs.

We have developed a Patch Object format for which the following holds:

- A PO is a valid ELF file.
- A PO utilizes DWARF information to describe types, variables, and functions requiring patching.
- A PO allows type transformations to be specified using a language defined by the DWARF standard.

Through the use of existing standards and well-structured ELF files utilizing a simple expression language for data patching, we aim to create patches that are easily examined (or modified) with existing tools. This easy compatibility with the existing binary tools and standards brings us to a very good point: why should patching not be a part of the ABI and of the standard toolchain? This does not necessarily have to be the precise format we use for Katana. Any such format that would become a standard, whether an actual standard or a *de facto* standard, should be well vetted by the community, but we argue that something like this should be included in the standard object types, along with relocatable objects, executable objects, shared libraries, and core dumps. Consider the situation. Relocatable objects containing new code and data which may be inserted at runtime are nothing new. This is the entire premise of the dynamic library. User-written functions which may have to run upon this code injection (in the case of patching data where the desired actions cannot be determined automatically) already exist as the `.init` and `.fini` sections. Because of this similarity between some of the functionality needed by patching and the functionality offered by dynamic libraries, some previous systems have performed patching by creating patches as dynamic libraries that contain not only the code and data to be patched but also the mechanism to perform the patching (Neamtiu et al., 2006) (Chen et al., 2007). We argue that this is an unnecessary mixing of data and logic and, further, that a patch that contains merely the information necessary to fix a running process and not the code to do so is more desirable. The code to apply the patch should live in one place on any given system, as most other executable content does. We do not embed Emacs within our text files, after all. Dynamic libraries and other relocatable, linkable objects do not contain code and data intended to overwrite data in an existing executable or process. Consider, however, that redefining certain symbols is only a slight twist on ordinary linker behaviour. Ordinary linker behaviour for global symbols is to fail if a symbol is defined more than once. Ordinary behaviour for weak symbols is to use a global definition if available or the weak symbol otherwise. When performing dynamic linking, generally the first appropriate symbol encountered in the chain of symbol tables is used. It is not a far difference to define the linkage rule that the symbol definition from the most recent patch takes precedence. Therefore, applying a patch consists of the following steps

1. Injecting appropriate sections of the patch into memory. This includes putting their contents into memory and performing relocations on these sections (but not on the rest of the in-memory process) so that they fit into their environment

2. Copying existing data to the appropriate regions of the newly mapped-in patch
3. Performing relocation on the entire in-memory process such that the symbols defined by the patch take precedence.

These steps are all such fundamental operations that they should become universally supported by the ABI and the toolchain.

On the other hand, note that the specification of a general patch format does not completely prescribe the patch application. From a standard patch format, a patcher is still free to make decisions such as when to patch safely and whether to patch functions by inserting trampolines in the old versions of the functions or by relocating all references to the function (we currently do the former in Katana, but may later transition to doing the latter).

Our Patch Object Format.

Our Patch Object (PO) format is an ELF-based format. Figure 3 shows the sections contained in a simple patch. `.text.new` and `.rodata.new` are of course the new code and supporting

```
Section Headers:
  [Nr] Name              Type
  [ 0]                   NULL
  [ 1] .strtab           STRTAB
  [ 2] .symtab           SYMTAB
  [ 3] .text.new         PROGBITS
  [ 4] .unsafe_functions LOUSER+1
  [ 5] .rodata.new       PROGBITS
  [ 6] .rela.text.new    RELA
  [ 7] .debug_info       PROGBITS
  [ 8] .debug_abbrev     PROGBITS
  [ 9] .debug_frame      PROGBITS
  [10] .rel.debug_info   REL
  [11] .rel.debug_frame  REL
```

Figure 3: Headers for the PO

constants to inject. `.rela.text.new` allows `.text.new` to be properly relocated after it is adjusted. While System V based systems use only relocation sections of type `SHT_REL`, we chose to use `SHT_RELA` in our patch objects because they make addends much easier to keep track of as we relocate from patched binary to patch object to patched process in memory. This is all really nothing new; storing ELF sections to be injected in-memory has been done before in other systems (Vanegue et al., 2009). What is new in our patch object is the inclusion of DWARF sections. The `.debug_info` section in an ordinary executable program contains a tree of DIEs (Debugging Information Entities) with information about every type, variable, and procedure in each compilation unit in the program. In a patch object, we store information only about the procedures and variables which have changed. This of course includes storing the type

information for changed variables. An example of the DWARF DIE information contained in a patch can be seen in Figure 4.

Note that we store considerably less information about each entity than is typically contained. This is so because we read most of the information from the DWARF and symbol table information of the executing process (unless it will have been stripped; then more information must be stored in the patch). This allows the patch to be more flexible as it does not require that all variables and procedures be located at exactly the addresses they were expected to be at when the patch was generated. This flexibility allows a single patch between versions *va* and *vb* to patch both an executable that was originally compiled to *va* and an executable that was patched from earlier versions to be equivalent to *va*. Ksplice, one of the few other patchers that operate solely at the binary level, does not have this capability (Arnold and Kaashoek, 2009). We will provide a mechanism for composing patches such that a patch from version *va* to *vb* may be composed with a patch from *vb* to *vc* to produce a patch from *va* to *vb*. Note that patch versioning is currently a work in progress and not fully implemented.

```
.debug_info

COMPILE_UNIT<header overall offset = 0>:
<0><   11> DW_TAG_compile_unit
    DW_AT_name                  main.c

LOCAL_SYMBOLS:
<1><   19> DW_TAG_subprogram
    DW_AT_name                  printThings
    DW_AT_low_pc                0x0
    DW_AT_high_pc               0x70
<1><   40> DW_TAG_structure_type
    DW_AT_name                  _Foo
    DW_AT_byte_size             16
    DW_AT_MIPS_fde              16
    DW_AT_sibling               <93>
<2><   55> DW_TAG_member
    DW_AT_name                  field1
<2><   63> DW_TAG_member
    DW_AT_name                  field_extra
<2><   76> DW_TAG_member
    DW_AT_name                  field2
<2><   84> DW_TAG_member
    DW_AT_name                  field3
<1><   93> DW_TAG_base_type
    DW_AT_name                  int
    DW_AT_byte_size             4
<1><   99> DW_TAG_variable
    DW_AT_name                  bar
    DW_AT_type                  <40>
```

Figure 4: DWARF DIEs in the PO

Most of the information in the DIE tree is concerned only with names or how to locate code within the patch object (high and low pc). Of special interest, however, is the `fde` attribute of the `DW_TAG_structure_type`. This attribute specifies an offset in the `.debug_frame` section of an FDE (Frame Description Entity). DWARF FDEs are designed for use in transforming one call frame into the previous call frame, and thereby walking up a call stack for either debugging purposes or exception-handling purposes (using the `.eh_frame` section). Transforming one call frame to another, however, is not such a different operation from transforming one structure to another version of the same structure. We have aided this use with an implementation of the DWARF virtual machine that defines several special register types (exploiting the fact that for the purposes of generality, DWARF registers are specified as LEB128 numbers, giving an unlimited number of registers). The DWARF register instructions contained in the FDE referenced in Figure 4 for copying `field1`, `field2`, and `field3` from the original version of a structure `_Foo` to a new version of `_Foo` that has gained the extra member `field_extra` in the middle of the existing fields would be represented as in Figure 5.

```
DW_CFA_register {CURR_TARG_NEW,0x4 bytes,0x0 off}
{CURR_TARG_OLD,0x4 bytes,0x0 off}
DW_CFA_register {CURR_TARG_NEW,0x4 bytes,0x8 off}
{CURR_TARG_OLD,0x4 bytes,0x4 off}
DW_CFA_register {CURR_TARG_NEW,0x4 bytes,0xc off}
{CURR_TARG_OLD,0x4 bytes,0x8 off}
```

Figure 5: FDE instructions for data patching

`CURR_TARG_NEW` and `CURR_TARG_OLD` are special symbolic values defined by the virtual machine. If we are patching the variable bar, then the `CURR_TARG_OLD` will be the old address of bar (its value in the symbol table), and `CURR_TARG_NEW` will be the new address bar is being relocated to. Our registers take advantage of the LEB128 encoding to hold a considerable amount of information in the register identifier. In the case seen above, the first byte identifies the class of the register (`CURR_TARG_NEW` or `CURR_TARG_OLD` in this example), the following word specifies the size of the storage addressed (this is included so that register assignments may copy an arbitrary number of bytes), and the final word specifies an offset from the address referred to by `CURR_TARG_(NEW—OLD)`.

6 RELATED WORK

There are several hot-patching systems preceding Katana. One of the most well-known is probably Ginseng (Neamtiu et al., 2006). Ginseng — and systems drawing inspiration from it such as Polus (Chen et al., 2007) — have successfully demonstrated patching of such important software as apache and sshd. These systems perform analysis of the differences between the original and the patched versions at the source code level. This introduces considerable (and we argue unnecessary) complexity and inability to deal well with some optimizations such as inlining and hand-written assembly. The complexity of analyzing the source code ties these systems to generally a single language (C in the case of both Ginseng and Polus). By contrast, Katana is language agnostic as it works at the level of the binary ABI, and although we have not yet demonstrated its doing so, it should

eventually be able to patch binaries compiled from any language, providing that the necessary symbol and relocation information is supplied. Ginseng also requires significant programmer interaction in annotating the code (Neamtiu, 2009) and requires compiling the code to use type-wrappers, allowing the patching of data types but at the cost of indirect access to them. The more programmer effort involved in generating a patch, the more likely the patch is to be incomplete or incorrect.

Motivated by many of the points in the above paragraph, the successful Ksplice system (Arnold and Kaashoek, 2009) patches at the binary level, as we do. We claim the following differences from and improvements over Ksplice.

- Ksplice operates on the kernel. As their paper states, most of their technique is not specific to the kernel, but there is no evidence that it has been implemented to function on userland programs. Katana operates on userland.
- Ksplice makes no attempt to patch data, relying entirely on programmer-written transformation functions when data types do change
- Ksplice patches are created as kernel modules. Ksplice does not provide a mechanism to perform operations, such as composition, on these patches.

To the best of our knowledge, Katana is the first system to utilize DWARF type information in patching.

Maintaining continuous availability, even in the absence of disruptive events like patches, is both a challenging technical exercise and the driving need for research on dependability, reliability, and fault tolereance (Zhou et al., 2007). Our work follows work focusing on enabling a software application to continue providing service or survive significant events like errors, exploits, and patches. This body of work includes research on dynamic kernel updates, software survivability, and software self-healing. However, other research areas also addressed the challenge of enabling software to adapt at runtime, e.g., the area of software evolution (e.g., (Stefano et al., 2004)).

The concept of crash-only software (Candea and Fox, 2003) advocates microrebooting: the procedure of retrofitting each component of a system with the ability to crash and reboot safely as the default mode of operation. Despite its appeal as a design principle, such an approach would be difficult to retrofit to legacy software. Although restarting a particular service or application is disruptive enough, rebooting the operating system itself multiplies this disruption. The need to avoid that kind of downtime helped drive the creation of frameworks like Loadable Kernel Modules for Linux, which allow for extending the kernel during runtime without a reboot. The ability to update the running kernel (as opposed to adding or removing modules) without rebooting was achieved at least ten years ago (Cesare, 1998) and recently rediscovered, albeit mostly for research, rather than commodity, kernels (sd and devik, 2001, Baumann et al., 2007, Soules et al., 2003)). Even so, dynamic updates of the kernel during runtime that don't require a reboot are difficult to apply to a commodity OS, although several efforts have been successful for the K42 experimental system (Soules et al., 2003, Baumann et al., 2005).

Software self-healing aims at ensuring continuous or increased availability for systems subjected to exploited vulnerabilities, either by automatically generating patches (Weimer et al., 2009, Sidiroglou et al., 2005) to gradually harden the application or by seeking to avoid a restart altogether by modifying certain runtime aspects (*e.g.,* the memory subsystem (Rinard et al., 2004), properties of the execution environment (Qin et al., 2005)), or selected control paths (Smirnov and Chiueh, 2005, Locasto et al.,

2007)) of the system in response to attacks. One major risk of employing self-healing in production environments is that the semantics of follow-on execution remains largely uncontrolled, although recent work in automatically correcting memory errors (Novark et al., 2008) seems to achieve fairly reliable results. Both automated responses and traditional patches can make it difficult for an administrator to understand the implications of a particular fix (Rinard, 2008).

7 CONCLUSION

We introduce a method for hot patching: a technique we believe to be a promising alternative to redundancy, ad hoc self–healing techniques, "patch and pray," or other approaches to dynamic software updates. Hot patching has the potential for aligning actual practices with acknowledged "best practices" relating to critical security or functionality updates. We hold that one major impediment to hot patching is the opaque nature of most patches (be it proprietary or open software), and our method of patching, along with the PO file format, is a first attempt at providing a basis for informed reasoning about the structure and implications of a patch.

We present a reasoned approach to making patching a part of the standard tool chain. We demonstrate a working binary userland patcher operating completely at the object level. Our system is, to our knowledge, the first to utilize DWARF type information to automate the transformation between old and new versions of a type. There yet remains much work to be done, and our future work involves support for patching multithreaded targets, better support for handling opaque types such as `void*`, and further development of patch versioning and the ability to perform operations on patch objects.

REFERENCES

Arnold, J. and Kaashoek, M. F. (2009). Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of EuroSys*.

Baumann, A., Appavoo, J., Wisniewski, R. W., Silva, D. D., Krieger, O., and Heiser, G. (2007). Reboots Are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly. In *Proceedings of the USENIX Annual Technical Conference*.

Baumann, A., Heiser, G., Appovoo, J., Silva, D. D., Krieger, O., Wisniewski, R., and Kerr, J. (2005). Providing Dynamic Update in an Operating System. In *Proceedings of the USENIX Annual Technical Conference*, pages 279–291.

Brown, A. and Patterson, D. A. (2002). Rewind, Repair, Replay: Three R's to dependability. In *ACM SIGOPS European Workshop*, Saint-Emilion, France.

Candea, G. and Fox, A. (2003). Crash-Only Software. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HOTOS-IX)*.

Cesare, S. (1998). Runtime Kernel kmem Patching. http://vx.netlux.org/lib/vsc07.html.

Chen, H., Yu, J., Chen, R., Zang, B., and Yew, P.-C. (2007). Polus: A powerful live updating system. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 271–281, Washington, DC, USA. IEEE Computer Society.

Ikebe,Takashi and Kawarasaki, Yasuro. (2006). http://pannus.sourceforge.net/.

Locasto, M. E., Stavrou, A., Cretu, G. F., and Keromytis, A. D. (2007). From STEM to SEAD: Speculative Execution for Automatic Defense. In *Proceedings of the USENIX Annual Technical Conference*, pages 219–232.

Neamtiu, I. (2009). Ginseng user's guide. http://www.cs.umd.edu/projects/PL/dsu/software.shtml. Contained in source distribution from the web page.

Neamtiu, I., Hicks, M., and Stoyle, G. (2006). Practical dynamic software updating for c. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation*, pages 72–83.

Novark, G., Berger, E. D., and Zorn, B. G. (2008). Exterminator: Automatically correcting memory errors with high probability. *Commun. ACM*, 51(12):87–95.

Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. (2005). Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)*.

Rinard, M., Cadar, C., Dumitran, D., Roy, D., Leu, T., and W Beebee, J. (2004). Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings Symposium on Operating Systems Design and Implementation (OSDI)*.

Rinard, M. C. (2008). Technical perspective patching program errors. *Commun. ACM*, 51(12):86–86.

sd and devik (2001). Linux on-the-fly Kernel Patching Without LKM. http://doc.bughunter.net/rootkit-backdoor/kernel-patching.html.

Sidiroglou, S., Locasto, M. E., Boyd, S. W., and Keromytis, A. D. (2005). Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*, pages 149–161.

Smirnov, A. and Chiueh, T. (2005). DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*.

Soules, C. A. N., Appavoo, J., Hui, K., Wisniewski, R. W., da Silva, D., Ganger, G. R., Krieger, O., Simon, M., Auslander, M., Ostrowski, M., Rosenburg, B., and Xenidis, J. (2003). System Support for Online Reconfiguration. In *Proceedings of the USENIX Annual Technical Conference*, pages 141–154.

Stefano, A. D., Pappalardo, G., and Tramontana, E. (2004). An infrastructure for runtime evolution of software systems. *Computers and Communications, IEEE Symposium on*, 2:1129–1135.

The ELF shell crew (2005). Embedded elf debugging : the middle head of cerberus. *Phrack Magazine*, 11(63).

Ukai, F. (2004). http://ukai.jp/Software/livepatch/.

Vanegue, J., de Medeiros, J. A., Bisolfati, E., Desnos, A., Figueredo, T., Garnier, T., Lesniak, R., Palencia, J., Roy, S., Soudan, S., Woloszyn, M., and Zabrocki, A. (2009). The eresi reverse engineering software interface. http://www.eresi-project.org/.

Weimer, W., Nguyen, T., Goues, C. L., and Forrest, S. (2009). Automatically Finding Patches Using Genetic Programming. In *International Conference on Software Engineering (ICSE)*.

Yamato, K. and Abe, T. (2009). A Runtime Code Modification Method for Application Programs. In *Proceedings of the Ottawa Linux Symposium*.

Zhou, Y., Marinov, D., Sanders, W., Zilles, C., d'Amorim, M., Lauterburg, S., Lefever, R. M., and Tucek, J. (2007). Delta Execution for Software Reliability. In *Proceedings of the Third Workshop on Hot Topics in System Dependability (HotDep'07)*.

NOTES

1. By which we mean manual, human-level reasoning, although applying automated reasoning methods is an interesting (and open) avenue of research.
2. For example, consider adding a new member to a C struct definition and an additional clause to the logic that processes it.
3. http://dwarfstd.org
4. In reality, the choice of offset is still limited by the platform's alignment requirements.
5. We note that saving this information about the post-relocation layout of the process does not weaken “randomization,” for the latter does not assume the attacker's ability to arbitrarily read process memory (in which case the address of required symbols are easily found by scanning it for their code or data patterns), but rather breaks hard-coding of these symbols' expected addresses.