

Katana: A Hot Patching Framework for ELF
Executables
Dartmouth Computer Science Technical
Report TR2009-657

Ashwin Ramaswamy¹, Sergey Bratus¹, Michael E. Locasto²,
and Sean W. Smith¹

¹Dartmouth College

²George Mason University

November 3, 2009

Abstract

Despite advances in software modularity, security, and reliability, 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 forego its benefits as too risky.

In this paper, we propose 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 propose a format, which we call a *Patch Object*, for encoding patches as a special type of ELF relocatable object file. Our tool, *Katana*, 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.

“Some reports, such as the case of the Conficker outbreak within Sheffield Hospital’s operating ward, suggest that even security-conscious environments may elect to forgo automated software patching, choosing to trade off vulnerability exposure for some perceived notion of platform stability...” – <http://mtc.sri.com/Conficker/>

1 Introduction

It is somewhat ironic that users and organizations 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 an extensive backup and failover infrastructure with rolling updates. 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), downtime is a foregone conclusion even as the software is released.

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 [6, 5] 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 likely wish to apply the patch and upgrade the process without actually restarting the program and losing state and time. This requirement serves as our motivation for *hot patching*.

1.1 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**.

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 or compatibility of the target with other software in any other way than applying the patch on the 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, reasoning¹ about its interaction with the rest of the system, and an understanding of the tradeoffs involved in applying it.

1.2 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:

- a. **expensive** - especially in medium-sized enterprises where the cost of a single server, gateway, or switch is high enough to outweigh the benefits of redundancy.
- b. **wasteful** - Redundant systems are typically passive bystanders, lying in wait for an active machine to initiate a failover.
- c. **requires 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).
- d. **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 fallback.

2 Katana Design: Tracking Object Dependencies

Katana leverages the typical Unix `Makefile` build mechanism to track file-level dependencies. Normally, after applying a source patch file and performing a top-level `make`, only those object files whose underlying sources have

¹By which we mean manual, human-level reasoning, although applying automated reasoning methods is an interesting (and open) avenue of research.

changed are rebuilt. Katana thus tracks object-level (.o) dependencies as follows. We first replicate all object files and the ELF executable from the existing source tree. We then apply the patch to the *original* source tree. At this point, only source files have been modified. Next, using the Linux kernel’s *inotify* mechanism², Katana sets up a notification on the original source tree, so that it knows when an object file is created or modified under the original tree. Finally, we perform the top-level `make` under this source and record all created/modified object files, along with the newly created executable.

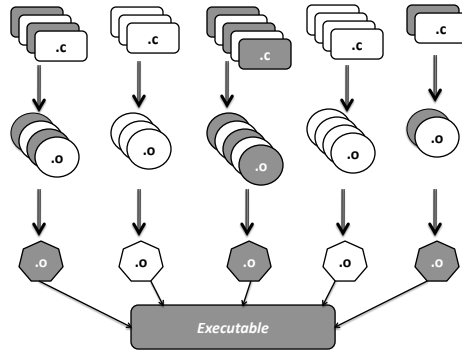


Figure 1: *An Example Code Base*. From the top: each source file creates a corresponding object file; multiple object files are combined into intermediate compilation units (CU); and multiple CUs are merged to form the executable. All shaded blocks indicate modified files.

Figure 1 illustrates how object files are modified by `make` as a result of source-level changes. Katana only considers objects that are closest to the source files and ignores all other intermediate object files and compilation units (CU). Hence, in Figure 1, Katana only records the shaded circle-objects along with the final executable.

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.*

²*inotify* allows the registration of filesystem triggers

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 The process of code patching 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. This list gives us the set of all functions that need to be patched within the executing application. We note that just because an object file has been modified does not mean *all* functions (at the source-level) within that object have necessarily been modified, but since it is not possible for us to determine which exact functions a patch modifies³, we resort to fully patching all functions within a modified object. Functions thus identified as needing patching 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 uses the replicated executable (from *before* the patch) to figure out the address of the symbol. If the symbol was provided by another object file, then the symbol table of the old executable contains this final address, and we update the PO accordingly, with this address as the patch target. Otherwise, if the symbol was dynamic (*i.e.*, present in a shared library such as `libc`), then the fixup value is the address of

³as local and weak symbols are not unique within an executable

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 replicated executable and compares the symbol name of each entry with the symbol name that requires relocation. If Katana finds a match, then the *PO* is updated with the corresponding symbol value. In our prototype, Katana is unable to handle calls to previously unused functions present in any shared libraries⁴

Finally, if the outstanding symbol's definition was not found within the replicated executable (either within the symbol table or 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 [1, 2, 3, 14]. 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 misprediction) of the `jmp` instruction by adding code in the old function which 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 make the process of rolling back a patch non-trivial.

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.⁵

Tracking down previously allocated data is nontrivial (one of the reasons why garbage collectors are interwoven with the language implementation).

⁴This would require creation of new *PLT* and *GOT* entries and either subsequent re-basing of the following segments of the executable, or creation of a new segment to allocate the extra entries. Although ELF rewriting systems like *ERESI* or *Diablo* show that such manipulations can be made practical, we chose not to complicate Katana with them.

⁵For example, consider adding a new member to a C struct definition and an additional clause to the logic that processes it.

Even after identifying the allocated chunks of memory, in the absence of some kind of a 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.

We first note that any code that acts on patch-modified data is already taken care of by Katana's code patching process. This is 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 DWARF⁶ 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 only need DWARF information while building the PO, all debugging symbols can be stripped from the executable during application deployment, if so desired. We 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, the parent type identifying the struct/union is inserted into the PO. Else, if the type name

⁶<http://dwarfstd.org>

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 if 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 leave the symbol resolution upto 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 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 if 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 automatically zeros all *new* member variables within structures.

Challenges Hot patching still faces a number of challenges, including dealing with multithreaded programs and address space randomization (which slight changes to the OS loader can help us overcome). Furthermore, dynamic updates require some knowledge of the program's execution state so that the application is quiescent with respect to the code and data being altered by Katana as it applies the Patch Object. We consider the program

to be in a safe state if all activation records are free of functions contained in the PO and all activation records are free of functions that (1) access any global or static symbols we identify during Katana’s *Data Traversal* stage and (2) do not define any local variables of modified types identified during our Type Discovery phase. Katana uses the `ptrace` interface to pause execution and query this state.

4 Related Work

The work most closely related to Katana focuses on enabling a software application to continue providing service or survive significant events like errors, exploits, and patches. The concept of crash-only software [6] advocates microbooting: 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 ability to update the running kernel (as opposed to adding or removing modules) without rebooting was achieved at least ten years ago [7] and recently rediscovered, albeit mostly for research, rather than commodity, kernels [10, 4, 12]). Finally, software self-healing aims at ensuring continuous or increased availability for systems subjected to exploited vulnerabilities, either by automatically generating patches [13, 11] to gradually harden the application or seeking to avoid a restart altogether by modifying certain runtime aspects (*e.g.*, the memory subsystem [9], properties of the execution environment [8]).

5 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 are first attempts at providing a basis for informed reasoning about the structure and implications of a patch.

References

- [1] <http://pannus.sourceforge.net/>.
- [2] <http://ukai.jp/Software/livepatch/>.
- [3] J. Arnold and M. F. Kaashoek. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of EuroSys*, 2009.
- [4] A. Baumann, J. Appavoo, R. W. Wisniewski, D. D. Silva, O. Krieger, and G. Heiser. Reboots Are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly. In *Proceedings of the USENIX Annual Technical Conference*, June 2007.
- [5] A. Brown and D. A. Patterson. Rewind, Repair, Replay: Three R's to dependability. In *10th ACM SIGOPS European Workshop*, Saint-Emilion, France, Sept. 2002.
- [6] G. Candea and A. Fox. Crash-Only Software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HOTOS-IX)*, May 2003.
- [7] S. Cesare. Runtime Kernel kmem Patching, 1998. <http://vx.netlux.org/lib/vsc07.html>.
- [8] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)*, 2005.
- [9] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings 6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.
- [10] sd. Linux on-the-fly Kernel Patching Without LKM. <http://doc.bughunter.net/rootkit-backdoor/kernel-patching.html>.
- [11] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*, pages 149–161, April 2005.

- [12] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. da Silva, G. R. Ganger, O. Krieger, M. Simon, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System Support for Online Reconfiguration. In *Proceedings of the USENIX Annual Technical Conference*, pages 141–154, 2003.
- [13] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically Finding Patches Using Genetic Programming. In *International Conference on Software Engineering (ICSE)*, 2009.
- [14] K. Yamato and T. Abe. A Runtime Code Modification Method for Application Programs. In *Proceedings of the Ottawa Linux Symposium*, 2009.