Repair from a chair: Computer repair as an untrusted cloud service

Lon Ingram*, Ivaylo Popov*, Srinath Setty*, and Michael Walfish* *UT Austin

1 Introduction

Today, when people need their computers repaired, their process is not very different from hiring someone to fix a television: they either bring the computer to a repair service [1, 3], or they call a technician (or family member) and ask for a house call. This process is inconvenient. It also risks the privacy of customers' data and the integrity of their systems: repair services have gained notoriety for stealing personal data from customers [22] or otherwise intruding on their privacy [7].

Remote desktops [2, 4, 5] avoid physical movement but still require that the customer make a choice. The customer can spend time and monitor the repairer (though many customers of repair services probably do not have the technical savvy to detect spurious actions in the first place). Or, the customer can save time and ignore the repairer, giving him carte blanche, just as if the computer were in the shop. Neither choice seems great.

The purpose of this paper is to articulate a new vision for repair. This vision is motivated by three trends:

- **Software repairs.** We surveyed retail repair services and learned that a large majority of repairs today involve software changes only (§2). Such repairs do not inherently require travel.
- **Virtual machines.** Many computers today, even desktops, include virtualization technology, which could allow customers to ship their computers electronically [9, 19] (by sending the virtual machine image) and enforce guarantees against the repairer (by implementing protections in the virtual machine monitor).
- **Outsourcing.** Service providers have long offered value-added services, such as desktop management and IT consulting. Lately, commodity computing has followed an analogous path, migrating to well-provisioned, off-site, partially anonymous service providers (often known as the *cloud*).

We call our vision *repair from a chair*: let a customer, at the press of a button, electronically ship a computer to a third party repairer whom the customer never meets; let the repairer be untrusted by the customer (meaning that the customer is protected against repairer error, whether accidental or intentional); and let the repair happen asynchronously. By *asynchronously*, we mean that the customer does not need to monitor the repair in real time.

Note that the context for this vision is *retail* repair, in contrast with much academic work on troubleshooting [8, 11, 14, 15, 17, 18, 20, 21, 23, 25–28]. There, an experienced system administrator is faced with a complex configuration issue (say a subtly wrong token in `httpd.conf`), and this person trusts the troubleshooter (or they are the same person). In our setting, however, users are inexperienced (so much so that they would be unable to use the above-cited tools), the problems are relatively easy for the troubleshooter (as indicated by our survey), and the customer does not fully trust the repairer. Thus, the technical challenges in our scenario are different (though the tools above would be useful to the repairer so are complementary to our work).

Our challenges are, first, to protect the *privacy* of the customer's data. For example, if the repairer needs to correct a misconfiguration in a virus checker, he should not be able to see private vacation photos. We also have to protect the *integrity* of the customer's system: if the repairer executes an invalid repair, a customer-side module should reject it or roll it back if the customer later discovers a problem. Last, we want to protect *availability*: the customer should be able to keep working during the repair. This requires a way to merge the repairer's changes with those of the customer.

A key building block for solving these problems comes from the rich literature on dependency tracking [8, 15, 17, 18, 20, 21, 23, 28] and, in particular, selective redo [11, 14], which we (ironically) use to protect against the repairer. We will also borrow other work, including virtual machine migration [6, 10, 19].

There are also new problems to solve: protecting customer privacy while allowing the repairer to work, statically validating repairs, merging the repairer's changes with the user's, dependency tracking across OS upgrades, coherently composing the aforementioned, and more. However, we do not yet have complete solutions. Rather, this paper's primary contributions are articulating both the vision and the research agenda that must be addressed to realize it. Secondary contributions are targeting retail repair (which implies a new model: easy repairs but untrusted repairers) and conducting an inquiry of current retail repair services, which we report next.

2 An inquiry into retail repair

The types of repairs that the architecture should accommodate and whether the vision even makes sense strongly depend on the nature of today's repairs. Accordingly, we now present a preliminary study of retail repair, deferring a proposed architecture to the next section.

Based on the tech support experience of one of the au-

On a scale from 1 to 5 (1 = never, 3 = sometimes, 5 = always),	μ	σ
How often do you repair a computer by ...		
... making a registry or other settings change?	3.7	0.6
... (re)installing an application program but not the entire OS?	3.2	1.0
... (re)installing a driver?	2.6	1.0
... (re)installing an automatic update from a software vendor?	2.6	1.4
... reinstalling the operating system?	2.7	0.7
... fixing or replacing a hard drive or other H/W component?	3.5	0.7
How often do you need access to the user's application data, such as photos, etc., in order to repair a computer?	1.5	0.7

Figure 1—Means and standard deviations of selected responses to Geek Squad survey in phase 2 ($n = 11$).

thors, we expected to hear that software issues are much more common than hardware issues and that, of the software fixes, application-level configuration changes are more common than OS re-installs. Our inquiry confirmed these impressions. It has three components: (1) telephone surveys of Best Buy Geek Squads in and around Austin; (2) incognito visits to six Apple Genius Bars; and (3) an analysis of 18 months of trouble tickets handled by UT Austin's Information Technology Services.

Our survey of the Geek Squads had two phases: one to gather a general sense of retail repair and the other to collect concrete data. In both phases, we cold-called Geek Squads in Best Buys in and around Austin and asked the agent who answered the telephone if he or she had time for a short survey. Of the ten stores that we called in the first phase, agents at seven agreed to participate.

Nearly all of the phase 1 respondents said that hardware problems were relatively uncommon, though they varied in their estimates of the exact proportion of hardware vs. software repairs. We also heard from every agent that they reinstall the OS in fewer than 20% of cases. Without exception, the agents named viruses as the most common customer issue. Typically, a Geek Squad agent is faced with a malware infection. Once the problem is diagnosed, the agent uses proprietary tools that attempt to clean the customer's computer, usually without reinstalling the operating system.

The second phase was more rigorous. It consisted of ten questions on a five point scale, and half of the surveys were conducted by someone who is not one of the authors and who did not know our hypotheses about which repairs are most common. Of the eighteen stores that we called, agents at eleven agreed to participate.

Figure 1 summarizes the second phase. Software repairs are apparently common. Indeed, looking at the individual responses, nine of eleven rated "making a registry or other settings change" at least as high (in frequency) as "replacing a hard drive or other hardware component". And application-level repairs appear more common than OS re-installs: nine of ten rated "making a registry or other settings change" at least as high as "reinstalling the operating system" (it was nine of ten, not eleven, because in one case we forgot to ask one of the questions). These findings are consistent with phase 1.

Next, we made incognito visits to six Apple Genius Bars (three in the Bay Area and one each in Austin, Fort Worth, and New York City). While the Geniuses were repairing our laptops, we asked them about their work. With the disclaimer that the plural of "anecdote" is not "data", we heard that the Geniuses deal with a mix of hardware and software, and that they usually fix software problems with a settings change, though they also reinstall the OS fairly often—most of which is consistent with the surveys. In contrast to the Geeks, the Geniuses were adamant about never seeing malware.

The findings above are obviously not scientific, but they are buttressed by a much larger dataset. We received 111,824 anonymized trouble tickets, for January 1, 2009 to June 8, 2010, from UT Austin's central Information Technology Services, which provides a range of IT support to 75,000 students, faculty, and staff. We wanted to know what percentage of repairs involve software versus hardware. We filtered the trouble tickets to remove issues unrelated to repair (such as network account administration and "how-to" questions). Of the remaining 15,572 tickets, the breakdown is 76% software, 24% hardware.

Our inquiry has been suggestive, but of course it is not conclusive or complete. First, despite the techs' best intentions, we cannot validate their diagnoses or remedies. Second, for the surveys and visits, the sample size was small. Third, all of the surveys and visits would ideally have been conducted by neutral investigators. A full-fledged empirical study is future work. For now, we wanted to understand retail repair enough to guide our architecture. Moreover, even if our estimates are inaccurate, as long as *some* significant fraction of repairs are software-only (and specifically application-only, for our first cut architecture), the architecture still has use.

3 Model and architecture

We now describe our model and architecture, deferring the accompanying challenges to the next section. Recall that in our retail repair setting, repairs are often easy, customers are technically inexperienced, and the repairer is not trusted. Thus, our model does not assume that the repairer acts correctly: accidentally or intentionally, he may try to make spurious repairs or violate the customer's privacy. An assumption of our model, in this section and in most of the next, is that repairs do not need to modify the kernel installed on the customer's system. This restriction simplifies some of the challenges, and it's supported by our empirical inquiry. On the other hand, the restriction is sizable, and we revisit it later (in §4.4).

Figure 2 depicts our envisioned architecture. The customer's computer is a virtual machine (VM), though in normal operation it need not run on a virtual machine

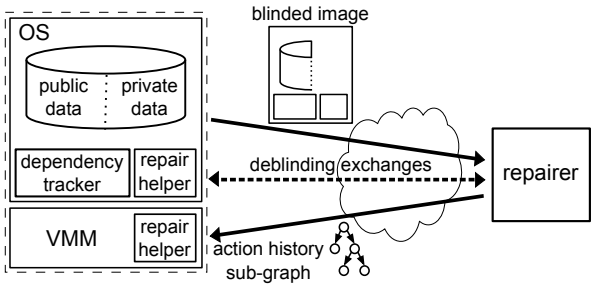

Figure 2—Architecture of repair from a chair.

monitor (VMM). The customer's files are partitioned into a public and a private part. (We address the feasibility of such a separation in §4.1.) The public part includes applications, system configuration, and kernel binaries and modules. The private part includes customer data.

While the customer works, the guest operating system tracks actions and dependencies [8, 11, 14, 15, 17, 18, 20, 21, 23, 28]; this tracking has several uses in our architecture. Our choice of encoding is an *action history graph*, a dependency graph proposed by RETRO [14] that stores the full contents of actions, not just causal links between them, making selective redo accurate and efficient. The customer's system also contains a *repair helper*, which is divided into a VMM piece and a guest OS piece.

When the customer requires a repair, the VMM piece of the repair helper uses VM migration technology [6, 10, 19] to ship to the repairer a copy of the file system with the private part blinded. A simple way to implement blinding is to not send the data; another way is to encrypt it. When the repairer wants access to private data, he makes requests to the customer, who can thus give informed consent (or decline). The customer can deblind individual files selectively and groups of them efficiently.

The repairer carries out the repair inside the image as it runs on a VMM. Note that the repairer's actions *should* generate an action history graph (since the customer-side OS tracks dependencies and since the repairer is not supposed to make kernel modifications). In fact, the repairer *must* encode any repair as an action history graph, since the OS piece of the customer's repair helper expects an action history sub-graph (rather than, say, a new image).

Given this sub-graph, the OS piece of the customer's repair helper (1) checks that the actions constitute valid repairs (for instance, that they do not modify the kernel or that they adhere to a defined list of applicable repairs); (2) leaves the door open to roll back the repairer's actions (if the repair is later found to be spurious) without discarding the customer's post-repair changes; and (3) merges the repairer's actions with any changes that the customer made while working during the repair.

Having outlined this architecture, we next outline some of the problems that must be solved to realize it.

4 Challenges and research

4.1 Privacy

To diagnose and fix problems, repairers sometimes need to see private files. For example, sensitive data may be in the configuration files involved in a repair [13]. For this reason and others, privacy raises challenges. To address them, we will walk through a series of tools below. These tools are not technically sophisticated, but the fact that they appear sufficient is itself interesting (at least to us). Indeed, the main contribution of the discussion below is to choose, from a wide design space, a concrete usage model and workable user experience for both repairer and customer. As a bonus, this model seems to admit a simple implementation.

Our goal here is a natural one: to limit disclosure to what is needed for the task at hand. Two observations put this goal within reach. First, most of a user's personal files are in well-known media-specific directories (e.g., `~/Photos`) that are rarely involved in computer problems, so these files are both easy to find and convenient to hide. Second, our personal experience in technical support is that most computer problems result from misconfigurations local to one file or a small set.

Thus, it makes sense for the repair helper to impose a default-deny policy on all potentially private data—including user-specific configuration files—but to support *deblinding exchanges* in which the repairer petitions the customer for access. When a repairer can read a directory but not the contents of its files, he can issue further deblinding requests. To minimize back and forth, inclusions and exclusions can be phrased recursively (e.g., "everything in `.firefox`, except for anything in `.firefox/favorites`"). The customer can then choose whether to grant these requests, possibly partially. We conjecture that we can empower even unsophisticated customers to arbitrate these decisions, using heuristic tools that clarify requests (e.g., "Careful: `.firefox/db` contains your passwords."). Validating this conjecture requires empirical inquiry (about the precise contents of repairs and about usability) and is future work.

Blinding and deblinding can be implemented simply by removing the contents of sensitive files from the file system image before shipping it to the repairer. To grant a deblinding request is to authorize one's repair helper to send a set of files to the repairer. An alternate implementation is to ship all the contents of the file system, suitably encrypted: eCryptfs [12], for example, offers per-file encryption and the ability to selectively deblind remotely. Note that, being in need of repair, the customer's OS may have a damaged network stack. Therefore, part of the deblinding machinery must be implemented in the VMM-level, not the OS-level, repair helper; identifying an appropriate interface between the two is future work.

While this discussion has so far concerned files, it extends to configuration databases like the Windows Registry, since we can map their hierarchy into the file system. For example, we can represent a registry key `\\HKEYS\A\B\C` in the blinded image as a file `\\Registry\HKEYS\A\B\C`.

The clean split that we presumed above is not always possible: the repairer must sometimes scan the whole disk (e.g., when searching for malware). For this case, we propose a restricted interface for executing diagnostic scripts under the *customer's* physical control. To fit into the broader deblinding scheme, such a script should produce a list of file names, say those whose contents match a regular expression. The repairer sets the regexps (e.g., a set of virus signatures), and the customer executes the script, optionally sanitizing the results before returning them to the repairer. One way to execute the script safely is to flash clone [24] the customer's machine into a disposable VM from which the VMM prevents data from exiting. Two research questions here are whether such an interface suffices for many retail repairs, and, if not, how to expand the interface and mechanism to let the repairer extract information beyond a list of file names.

Of course, the approach described here is not perfect. On a technical level, it does not strictly limit the flow of private data. Yet, we would guess that leaks of private data to locations that are repairer-visible *and* predictable will be rare (we address malicious repairs in §4.3), and in this retail setting (vs. a mission-critical one), we can probably tolerate an imperfect approach. On a social level, the repairer could entice, needle, or cajole the customer into deblinding a lot. This problem can be mitigated (e.g., with heuristics alerting the customer to social engineering) but not eliminated: it is inherent in any mechanism that gives final authority to humans.

4.2 Availability and consistency

We want to allow the customer to keep working during the repair. But then how can we merge the repairer's changes with any that the customer made since shipping a copy of the computer to the repairer? At a high level, we can accomplish the merge by *grafting* the repairer's changes into the customer-side action history graph.

That is, when the repair helper receives the repairer's action history sub-graph, it creates a dummy action that logically happens at the time when the customer's system shipped to the repairer. After a repair, the dummy action is always executed, and its redo action installs into the graph all of the repairer's actions (and their dependencies). By modifying the dependency tracker to store a vector clock for each event, rather than a timestamp, we impose the natural partial order on events. The repair helper can thus execute a roll-forward loop, borrowed from RETRO [14], to apply the customer's and the repairer's changes, while flagging conflicting changes.

Note that the action history graph's fine-grained dependencies are critical. With coarse-grained dependencies, we would have many more conflicts between repairer and customer. Of course conflicts are still possible, but our scenario is not inherently to blame: conflicts exist in any undo/redo system. Our scenario likely exacerbates the problem, but the extent to which it does so is another empirical question for us to investigate.

4.3 Protecting against spurious repairs

We would like to protect the customer's computer against spurious repairs. While a perfect solution to this problem seems impossible even in theory, our architecture could protect a customer far more than today's repair model does, if several challenges are addressed.

First, a customer (or subsequent repairer) should be able to roll back a repair, perhaps weeks later (if the customer learns that the repairer was malicious or if a subsequent repairer finds a spurious repair). The challenge is retaining a customer's updates. To address this challenge, we can use selective redo [11, 14]. A key mechanism here is the action history graph, which will hold the grafted-in repairer changes (§4.2). A subtlety is that the repairer's actions might be most naturally phrased as undoing prior customer actions (if the repairer received a sanitized action history graph from the customer). But such undo actions could be encoded in the action history graph, with natural extensions to RETRO [14].

Second, to preserve the integrity of the repair helper and the dependency tracker, the repair must not modify the guest kernel or its configuration. Since protecting the disk blocks that house kernel binaries and related configurations is straightforward, the challenge here is defending against *indirect* malicious repairs (e.g., installing a program that loads a module that subverts the repair helper). One response is to create a permission model that is more fine-grained than that of today's machines. We would need to create a user, `repairer`, with permission to change the configuration state of a machine but not to load kernel modules or modify kernel state on disk. Then, during replay, the repair helper must enforce that the repairer's actions happen as `repairer`.

Third, we would like the repair helper to reject invalid repairs up front. While a general solution here seems impossible, heuristics should be useful: static analysis of the action history graph may be possible, and prior work applies statistical techniques to stored event traces and a representation of the local machine to automatically answer questions like, "How do I do task X?" [16] or "What is the root cause of symptom Y?" [25, 27]. We need to adjust the techniques, as our questions (e.g., "Is this change a configuration-specific version of a known repair?", "Is this change within the bounds of typical repairs?") are

easier, but our setting is more adversarial.

Apart from the heuristics above (which will never be perfect—a repair could be, say, an insidious spy module that later leaks customer data by HTTP POSTing it), we can imagine a business approach with a technical component. If repairers are expected to sign individual actions, then malicious applications could be reliably traced to them, making it possible to take legal action. Given that world, the officers of retail repair services would presumably restrict their employees by pre-signing only a fixed set of repairs. How large must this set be? We conjecture, and must validate, that it would not be prohibitive.

Ultimately, none of the above can be error-free. Our hope instead is to protect customers far more than today, by combining selective redo, guaranteed rejection of a large class of spurious repairs, heuristic detection of anomalous repairs, and signed repairs. This may sound like a lot of functionality, but each piece is modular and layered on a common primitive—the action history graph—that is part of the architecture anyway.

4.4 Repairs that modify the kernel

We now briefly revisit our kernel-repair-not-needed assumption from §3 and mention some challenges raised by a series of increasingly intrusive repairs. What makes the problem tractable is that in our retail setting, kernel modifications should be only known kernel upgrades.

First, consider kernel upgrades that leave the repair helper's interfaces unmodified. Then, the repair helper can conceivably authenticate the upgrade, as that upgrade ought to be signed by the OS vendor (who also presumably supplied the repair helper). The technical challenge, given our architecture, is encoding such an upgrade (and its signature) as an identifiable unit in the action history graph. Second, if the kernel upgrade does modify the repair helper interface—say by changing the format of the action history graph—then supporting undo/redo is an added challenge. The reason is that the mid-repair changes (including the upgrade itself) and the post-repair changes will be encoded two different ways.

Last, we want to let the repairer provide a new OS image when a damaged OS does not boot (or whenever else the repairer prescribes an OS re-install). The VMM repair helper, then, must expose an interface to the customer, verify the new OS image, and optionally connect the reinstall action to the prior action history graph.

5 Conclusion

There are obviously a lot of unknowns here, and our future work is to answer the questions that we raised and others that we did not have space to treat, such as: How can we repair the VMM? How can we apply these ideas to synchronous repair (where the customer is online during the repair)? Do spurious repairs happen often? Is asynchronous operation truly important? If the answers to the latter two are negative, can we swap dependency tracking for a lighter-weight mechanism? Whatever form our system ultimately takes, we believe that the vision itself and the questions that it raises are worth pursuing.

Acknowledgments

We thank Donna Ingram for help conducting the Geek Squad survey; Aaron Reiser of UT Information Technology Services for the anonymized trouble ticket dataset; and Lorenzo Alvisi, Andrew Blumberg, Russ Cox, Josh Leners, Emmett Witchel, and Nickolai Zeldovich for conversations and comments that significantly improved the draft. This work was supported by AFOSR grant FA9550-10-1-0073, ONR grant N00014-09-10757, and NSF grant 1040083.

References

[1] Apple Genius Bar. `http://www.apple.com/retail/geniusbar`.

[2] Fog Creek Copilot. `http://www.copilot.com/`.

[3] Geek Squad. `http://www.geeksquad.com/`.

[4] GoToMyPC. `http://www.gotomypc.com/`.

[5] TightVNC Software. `http://www.tightvnc.com/`.

[6] VMware vMotion. `http://www.vmware.com/products/vmotion/`.

[7] Ars Technica. Geek squad guy caught playing peep squad. `http://arstechnica.com/old/content/2007/04/geek-squad-guy-caught-playing-peep-squad.ars`, Apr. 2007.

[8] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.

[9] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. Lam. The Collective: A cache-based system management architecture. In *NSDI*, 2005.

[10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[11] A. Goel, K. Po, K. Farhadi, Z. Li, and E. D. Lara. The Taser intrusion recovery system. In *SOSP*, 2005.

[12] M. A. Halcrow. eCryptfs: An enterprise-class cryptographic filesystem for Linux. In *Proc. Ottawa Linux Symposium*, 2005.

[13] Q. Huang, H. J. Wang, and N. Borisov. Privacy-preserving friends troubleshooting network. In *NDSS*, 2005.

[14] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In *OSDI*, 2010.

[15] S. T. King and P. M. Chen. Backtracking intrusions. In *SOSP*, 2003.

[16] N. Kushman and D. Katabi. Enabling configuration-independent automation by non-expert users. In *OSDI*, 2010.

[17] K.-K. Muniswamy-Reddy and D. A. Holland. Causality-based versioning. *Transactions on Storage*, 5(4):1–28, 2009.

[18] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *USENIX Technical*, 2006.

[19] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *OSDI*, 2002.

[20] S. Shah, C. A. N. Soules, G. R. Ganger, and B. D. Noble. Using provenance to aid in personal file search. In *USENIX Technical*, 2007.

[21] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: Improving configuration management with operating system causality analysis. In *SOSP*, 2007.

[22] The Consumerist. Computer techs caught overcharging, lying, and snooping through your personal files. `http://bit.ly/dSAkNA`, Aug. 2008.

[23] C. Verbowski, E. Kıcıman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev. Flight Data Recorder: Always-on tracing and scalable analysis of persistent state interactions to improve systems and security management. In *OSDI*, 2006.

[24] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *SOSP*, 2005.

[25] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI*, 2004.

[26] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI*, 2004.

[27] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. In *EuroSys*, 2006.

[28] N. Zhu and T. Chiueh. Design, implementation, and evaluation of repairable file service. In *DSN*, 2003.