Repairing COTS Router Firmware without Access to Source Code or Test Suites: A Case Study in Evolutionary Software Repair

Eric Schulte
GrammaTech, Inc.
531 Esty Street
Ithaca, New York 14850
eschulte@grammatech.com

Westley Weimer
Dept. of Computer Science
University of Virginia
Charlottesville, VA 22904
weimer@cs.virginia.edu

Stephanie Forrest
Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131
forrest@cs.unm.edu

ABSTRACT

The speed with which newly discovered software vulnerabilities are patched is a critical factor in mitigating the harm caused by subsequent exploits. Unfortunately, software vendors are often slow or unwilling to patch vulnerabilities, especially in embedded systems which frequently have no mechanism for updating factory-installed firmware. The situation is particularly dire for *commercial off the shelf* (COTS) software users, who lack source code and are wholly dependent on patches released by the vendor.

We propose a solution in which the vulnerabilities drive an automated evolutionary computation repair process capable of directly patching embedded systems firmware. Our approach does not require access to source code, regression tests, or any participation from the software vendor. Instead, we present an interactive evolutionary algorithm that searches for patches that resolve target vulnerabilities while relying heavily on post-evolution difference minimization to remove most regressions. Extensions to prior work in evolutionary program repair include: repairing vulnerabilities in COTS router firmware; handling stripped MIPS executables; operating without fault localization information; operating without a regression test suite; and incorporating user interaction into the evolutionary repair process.

We demonstrate this method by repairing two well-known vulnerabilities in version 4 of NETGEAR's WNDR3700 wireless router *before* NETGEAR released patches publicly for the vulnerabilities. Without fault localization we are able to find repair edits that are not located on execution traces. Without the advantage of regression tests to guide the search, we find that 80% of repairs of the example vulnerabilities retain program functionality after minimization. With minimal user interaction to demonstrate required functionality, 100% of the proposed repairs were able to address the vulnerabilities while retaining required functionality.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

1. INTRODUCTION

Embedded devices handle private data, operate heavy machinery, and run continuously while communicating over the Internet. End users are unable to read or write the software controlling these devices, or to patch known vulnerabilities. As embedded systems become increasingly ubiquitous, techniques enabling users to customize and protect the software running their devices will become increasingly important, *cf.* the *Internet of things* [3].

Router bugs are an important class of embedded system vulnerabilities, ranging from the bug in CISCO's IOS, which caused outages in nearly every country worldwide [42], to security vulnerabilities in home routers such as recent examples in NEGEAR [9], and D-Link [11]. Unfortunately major software vendors commonly delay releasing patches to security vulnerabilities. In a study of high and medium risk vulnerabilities in Microsoft and Apple products between 2002 and 2008, for example, about 10% of vulnerabilities were found to be still *un*-patched 150 days after disclosure, and on any given date from around 10 to over 20 disclosed vulnerabilities were public and un-patched for Microsoft and Apple respectively [14].

Rather than waiting for vendor-delivered patches, we propose a technique for users to repair vulnerabilities automatically, even when developer source code and test suites are not available. A user-produced patch could be installed temporarily for internal protection, redistributed with the exploit (reporting an exploit with a patch in hand has been shown to reduce the total number of attacks [2]), or sent to the software vendor to reduce development time for the official patch [40].

In recent years, a variety of automated methods for program repair have successfully repaired defects in real-world software (e.g., [32, 24, 20, 30]). Automated repair methods based on evolutionary computation (EC) have also repaired defects directly in x86 and ARM ELF files, without access to program source code [34, 36]. This prior work, however, relies on a regression test suite to define the required functionality, or informal specification, of the program under repair, and on fault localization information to guide the genetic operations. Here we consider a setting in which none of source code, test suites, or fault localization information is available, and there is no cooperation from the vendor.

We demonstrate our technique by patching multiple security vulnerabilities in the popular NETGEAR WNDR3700 wireless router, which at the time of writing NETGEAR has

not publicly addressed. Although previous EC program repair techniques explicitly require access to a regression test suite, we explore the feasibility of performing repairs without any test suite and find that for our demonstration vulnerabilities, regression test suites are most often not necessary. In addition, we find that multiple vulnerabilities can be repaired in a single evolutionary repair run.

The main contributions of this paper are as follows.

- EC repair without a regression test suite
- EC repair without fault localization
- EC repair that leverages user interaction
- EC repair in embedded firmware
- EC repair of a current real-world unpatched exploitable vulnerability
- Iterative EC repair of multiple vulnerabilities in a single repair run

To encourage reproducible research [6, 28] and to allow others to patch future vulnerabilities, we have published a companion open source repository.[1] It contains the instructions, source code, and tooling needed to extract, execute and repair the binary NETGEAR router image vulnerabilities, as well as the data used to generate the analyses and figures reported in this paper. In this work we aspire to empower users to patch important vulnerabilities quickly and researchers to release patches simultaneously with exploit announcements.

The remainder of the paper reviews two recent exploits of NETGEAR WNDR3700 (§2); details the extraction and execution of the NETGEAR firmware in a virtualized sandbox (§3.1); describes the automated program repair technique (§3.2 and §3.3); evaluates effectiveness and quality of repairs (§4); summarizes related work (§5); and discusses implications and limitations (§6).

2. DESCRIPTION OF EXPLOITS

We address two current exploits in version 4 of the NETGEAR WNDR3700 wireless router. The popularity of this router implies that vulnerable systems are currently widespread. For example, the *shodan*[2] device search engine returned hundreds of vulnerable publicly accessible WNDR3700 routers at the time of writing. Both exploits exist in the router's internal web server in a binary executable named `net-cgi`, and both are related to how `net-cgi` handles authentication [9].

The vendor-deployed binary is insecure in at least two ways:

1. Any URI starting with the string "`BRS`" bypasses authentication.
2. Any URI including the substrings "`unauth.cgi`" or "`securityquestions.cgi`" bypass authentication. This applies even to requests of the form http://router/page.html?foo=unauth.cgi, meaning that the vulnerability effectively applies to all internal webpages. Many administrative pages start with the "`BRS`" string, providing attackers with access to personal information such as users passwords. By accessing the page http://router/BRS_02_genieHelp.html, attackers can disable authentication completely in a way that persists across reboots.

[1] https://github.com/eschulte/netgear-repair

[2] http://www.shodanhq.com/search?q=wndr3700v4+http

3. AUTOMATED REPAIR METHOD

Figure 1 gives a high-level overview of the repair technique, which consists of three stages:

1. Extract the binary executable from the firmware and reproduce the exploit (§3.1).
2. Use EC to search for repairs by applying random mutations and crossover to the embedded stripped (without symbols or section tables) MIPS ELF binary (§3.2).
3. Interactively construct test cases and return to (2) as needed to improve the quality of unsatisfactory candidate repairs (§3.3).

The first step in repairing the `net-cgi` executable is to extract the vulnerable executable and the router file system from the firmware image distributed by NETGEAR. Using these we construct a test harness that can exercise the vulnerabilities in `net-cgi`. This test harness is used by the repair algorithm to evaluate candidate repairs and to identify when repairs to the vulnerabilities have been found.

3.1 Firmware Extraction and Virtualization

NETGEAR distributes firmware with a full system image for the WNDR3700 router, which includes the router file system that has the vulnerable `net-cgi` executable. The file system was extracted using the `binwalk`[3] firmware extraction tool, which scans the binary data in the raw monolothic firmware file, searching for signatures identifying embedded data sections, including a `squashfs` [26] section that holds the router's file system.

The router runs on a big-endian MIPS architecture, requiring emulation on most desktop system to safely reproduce the exploit and evaluate candidate repairs. We used the QEMU system emulator [4] to emulate the MIPS architecture. The extracted router file system is first copied into the emulated MIPS Linux system. Then a number of special directories (e.g., `/proc/`, `/dev/` etc.) are mounted inside the extracted file system and bound to the corresponding directories on the virtual machine. At this point, commands can be executed in an environment that closely approximates the execution environment of the NETGEAR router by using the `chroot` command to confine executable access to within the extracted NETGEAR file system. This also effectively *sandboxes* all trial executions. Additional details are given in http://eschulte.github.io/netgear-repair/INSTRUCTIONS.html.

Using this system the entire NETGEAR router can be run under virtualization. In particular, the router's web interface can be accessed either using an external web browser or the `net-cgi` executable can be called directly from the command line.

3.2 Automated Program Repair and ELF Files

We use EC methods [12, 24, 25, 15] to search for small changes to existing programs that eliminate undesired buggy

[3] http://binwalk.org

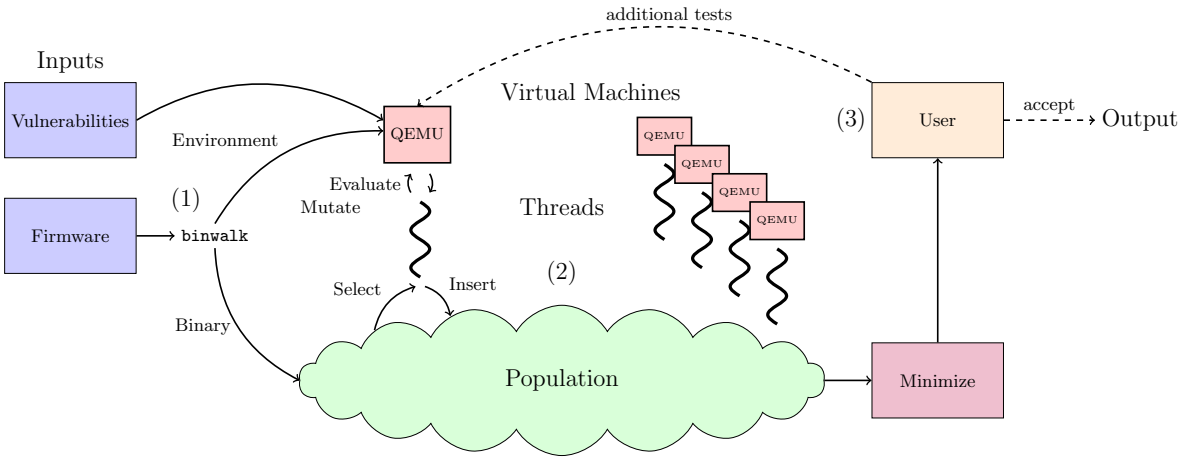

Figure 1: Three-stage automated evolutionary repair technique: (1) the vulnerable binary is extracted from the vendor-supplied firmware; (2) EC techniques are used to find versions of the binary which resolve the target vulnerabilities; (3) the user interactively adds test cases protecting broken functionality and returns to step 2 until an acceptable repair is found.

behavior. This process typically requires access to the source code of the original program, which is first transformed into an abstract syntax tree and then iteratively modified using *mutation* and *crossover*, guided by fault localization, to generate program variants. Each variant would then be evaluated in a process called fitness evaluation by running it against the program's existing regression test suite and at least one additional test that demonstrates the undesired behavior.

The repair algorithm constructs a population of program variants, each with one or more random mutations. This population is evolved through an iterated process of evaluation, selection, mutation, and crossover (pseudo-code provided in Figure 3) until a version of the original program is found that repairs the bug. "Repair" in this context is defined to mean that it avoids the buggy behavior and does not break required functionality. Execution traces collected during program execution are used as a form of *fault localization* to bias random mutations towards the parts of the program most likely to contain the bug.

We modify this repair algorithm in several ways to address the unique scenario of a user repairing a faulty binary COTS executable (§3.2.1), without access to a regression test suite (§3.3), and without the fault localization optimization (§4.2.2).

3.2.1 Challenge: Mutating Stripped Binaries

Executable programs for Unix and embedded system are commonly distributed as ELF (Executable and Linking Format) [7] files. Each ELF file contains a number of headers and tables containing administrative data, and sections holding program code and data. The three main administrative elements of an ELF file are the ELF header, the section table and the program table (see Figure 2). The ELF header points to the section table and the program table, the section table holds information on the layout of sections in the ELF file on disk, and the program table holds information on how to copy sections from disk into memory for program execution.

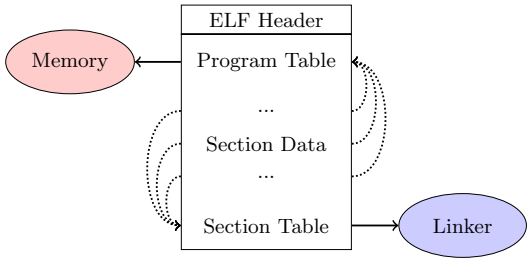

Figure 2: Sections and their uses in an Executable and Linking Format (ELF) file.

Although the majority of ELF files include all three of the elements shown in Figure 2, only the ELF Header is guaranteed to exist in all cases. In *executable* ELF files the program table is also required, and similarly, in *linkable* files the section table is required.

We extend previous work that repaired unstripped Intel and ARM files [36]. The ELF file is modfied by similar mutation and crossover operations, but in this case `net-cgi` does not include key information on which the earlier work relied, namely the section table and section name string table. This information was used to locate the `.text` section of the ELF file where program code is normally stored. The data in the `.text` section were then coerced into a linear array of assembly instructions (the *genome*) on which mutation and crossover operated. Our work removes this dependence by concatenating the data of every section in the program table that has a loadable (`PT_LOAD`) type to produce the genome. The genome thus includes all sections whose data are loaded into memory during program execution including both code and data.

Mutation operations must change program code without corrupting the structure of the file or breaking the many memory addresses hard coded into the program (e.g., as destinations for jumps, loads, or stores). In general, it is impossible to distinguish between an integer literal and an address in program code and data, so our mutation operations

are designed to preserve operand absolute sizes and offsets within the ELF program data. In addition, the preservation of absolute size ensures that the modified ELF file may directly replace the original in the firmware image. These requirements are more easily met on the MIPS RISC architecture because every argumented assembly instruction is exactly one word long [17].

The preservation of offsets allows the use of "Single point crossover" to recombine two ELF files. An offset in the program is selected, then bytes from one file are taken up to that offset and bytes from the other file taken after that offset. This form of crossover works especially well because all ELF files are *homologous*, i.e. they will have similar total length and similar contents at any given offset. Single point crossover has previously been shown effective for the evolution of homologous machine code [31]. The mutation and crossover operations used to modify stripped MIPS ELF files are shown in Figure 3.

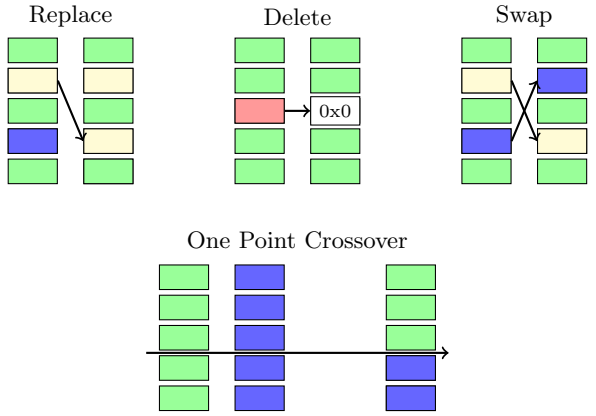

Figure 3: Mutation and Crossover operations for stripped MIPS ELF files. The program data are represented as a fixed length array of single-word sections. These operators change these sections maintaining length and offset in the array.

3.3 Interactive Regression Testing

Our approach to program repair relies on the ability to assess the validity of any candidate repair. The program transformations do not take into account or preserve the semantics of the program. They are more likely to create new bugs or vulnerabilities than they are to repair undesired behavior, and an evaluation scheme is required to distinguish between these cases.

Instead of relying on a pre-existing regression test suite, we assume only that a demonstration of the exploit provides a single available test. By mutating programs without the safety net of a regression test suite, the evolved "repairs" often introduce significant regressions. However, by applying a strict minimization process after the primary repair is evolved, these regressions are usually removed (§4.2.3). The minimization reduces the difference between the evolved repair and the original program to as few edits as possible using Delta Debugging [41]. The final phase of the repair algorithm asks the user to identify any regressions that remain after the Delta Debugging step through interactive use of the modified binary, e.g. through a web interface. If any failures are found, the user must manually write a regression test encoding the steps of the interactive process which led to the identification of the regression. For example after interactively identifying a broken web page, a user would write a regression test script which first downloads the broken page's URI, then checks for a successful download and for desired properties in the downloaded file, such as the presence of required components of the page. High-level pseudocode for the repair algorithm is show in Figure 4.

Our method is thus an interactive repair process in which the algorithm searches for a patch that passes every available test (starting with only the exploit), and then minimizes it using Delta Debugging. In a third step, the user evaluates its suitability. If the repair is accepted, the process terminates. Otherwise, the user supplies a new regression test that the repair fails (a witness to its unsuitability) and the process repeats. In §4 we find that 80% of our attempts to repair the NETGEAR WNDR3700 vulnerabilities did not require any user-written regression tests.

Input: Vulnerable Program, original : ELF
Input: Exploit Tests, vulnerabilities : $[ELF \rightarrow Fitness]$
Input: Interactive Check, goodEnough : $ELF \rightarrow [ELF \rightarrow Fitness]$

Output: Patched version of Program
1: **let** $new \leftarrow$ null
2: **let** $fitness \leftarrow$ null
3: **let** $suite \leftarrow$ vulnerabilities
4: **repeat**
5: **let** full $\leftarrow$ evolutionarySubroutine(original, suite)
6: $new \leftarrow$ minimize()
7: **let** $newRegressionTests \leftarrow$ goodEnough(new)
8: suite $\leftarrow$ suite + +newRegressionTests
9: **until** $length$(newRegressionTests) $\equiv 0$
10: **return** new

Figure 4: High-level Pseudocode for interactive evolutionary repair algorithm.

The `evolutionarySubroutine` in Figure 4 is organized similarly to previous work [24], but it uses a *steady state* evolutionary computational algorithm [27] for reduced memory usage, ease of parallelization of fitness evaluation, and recognizes the nearness of the original program to likely valid repairs [35, §2.2.2]. Figure 5 gives the high-level pseudocode. Note that every time the user rejects the solution returned by `evolutionarySubroutine`, the evolved and minimized solution is discarded and a new population is generated by recopying the original in `evolutionarySubroutine`. Most applications of EC begin with a randomly generated population, but we begin with a population of copies of the original. This is because the original program is in fact a highly engineered solution to the program fitness landscape and likely to lay close to acceptable repairs. This algorithmic choice acknowledges the fitness of the original program, and for this reason gives it primacy over the evolved solutions of previous iterations (which may well have evolved into fitness valleys as in run 8 Table 1).

4. REPAIRING THE NETGEAR VULNERABILITIES

We first describe the experimental setup used to test the repair technique on the NETGEAR WNDR3700 exploit (§4.1). We then analyze the results of ten repair attempts (§4.2).

```
Input:  Vulnerable Program, original : ELF
Input:  Test Suite, suite : [ELF → Fitness]
Parameters:  populationSize, tournamentSize, crossRate
Output:  Patched version of Program
 1: let fitness ← evaluate(original, suite)
 2: let pop ← populationSize copies of ⟨original, fitness⟩
 3: repeat
 4:    if Random() < CrossRate then
 5:       let p1 ← tournament(pop, tounamentSize, +)
 6:       let p2 ← tournament(pop, tounamentSize, +)
 7:       let p ← crossover(p1, p2)
 8:    else
 9:       p ← tournament(pop, tounamentSize, +)
10:    end if
11:    let p′ ← Mutate(p)
12:    let fitness ← evaluate(suite, p′)
13:    incorporate(pop, ⟨p′, Fitness(Run(p′))⟩)
14:    if length(pop) > maxPopulationSize then
15:       evict(pop, tournament(pop, tounamentSize, −))
16:    end if
17: until fitness > length(suite)
18: return p′
```

Figure 5: High-level Pseudocode for the steady state parallel evolutionary repair subroutine.

4.1 Methodology

All repairs were performed on a server-class machine with 32 physical Intel Xeon 2.60GHz cores, Hyper-Threading and 120 GB of Memory. We used a test harness to assess the fitness of each program variant (§4.1.1) and report parameters used in the experiments (§4.1.2).

4.1.1 Fitness Evaluation

We used 32 QEMU virtual machines, each running Debian Linux with the NETGEAR router firmware environment available inside of a `chroot`. The repair algorithm uses 32 threads for parallel fitness evaluation. Each thread is paired with a single QEMU VM on which it tests fitness.

The test framework includes both a host and a guest test script. The host script runs on the server performing repair and the guest script runs in a MIPS virtual machine. The host script copies a variant of the `net-cgi` executable to the guest VM where the guest test script executes `net-cgi` the command line and reports a result of Pass, Fail, or Error for each test. These values are then used to calculate the variant's scalar fitness.

Pass indicates that the program completed successfully and produced the correct result, Fail indicates that the program completed successfully but produced an incorrect result, and Error indicates that the program execution did not complete successfully due to early termination (e.g., because of a segfault) or by a non-zero "errno" exit value.

4.1.2 Repair Parameters

Our algorithm uses the following parameters. The maximum population size is 512 individuals, selection is performed using a tournament size of two. When the population overflows the maximum population size, an individual is selected for eviction using tournament selection in reverse. Newly generated individuals undergo crossover two-thirds of the time.

These parameters differ significantly from those used in previous evolutionary computation (EC) repair algorithms (e.g., [12, 15, 25]). Specifically, we use larger populations (512 instead of 40 individuals), running for many more fitness evaluations (≤100,000 instead of ≤400). The parameters used here are in line with those used in other EC publications given the size of the `net-cgi` binary, and they help compensate for the lack of fault localization information.

The increased memory required by the larger population size is offset by the use of a steady-state [27] EC algorithm, and the increased computational demand of the greater number of fitness evaluations is offset by parallelization of fitness evaluation.

4.2 Experimental Results

We report results for the time typically taken to generate a repair (§4.2.1), the effect of eliminating fault localization (§4.2.2), and the impact of the minimization process (§4.2.3), both with respect to the size of the repair in terms of byte difference from the original and in terms of the fitness improvement. Finally we demonstrate how multiple repairs can be discovered iteratively by the repair process (§4.2.4).

4.2.1 Repair Runtime

In 8 of the 10 runs of the algorithm (with random restarts), the three exploit tests alone were sufficient to generate a satisfactory repair (determined using a withheld regression test suite hand-written by the authors[4]), and the third phase of user-generated tests was not required.

In these cases the repair process took an average of 36,000 total fitness evaluations requiring on average 86.6 minutes to find a repair using 32 virtual machines for parallelized fitness evaluation.

4.2.2 Repair without Fault Localization

In the NETGEAR scenario, we do not have access to fault localization information. Although commonly required, fault localization information may sometimes *over-constrain* the search operators (mutation and crossover) [37], preventing the discovery of valid repairs.

One of the NETGEAR vulnerabilities exemplifies this issue. As shown in Figure 6, fault localization might have prevented the repair process from succeeding. The figure shows that many of the program edit locations for successful repairs were not visited by the execution trace. In fact, only 2 of these 22 program edit locations were within 3 instructions of the execution traces. In fact, one of the edit locations was in the `.rodata` section of the binary which could never appear in an execution trace. The `.rodata` section typically holds read-only data such as static variables. Such non-code sections of the executable were only mutated because of the lack of section names as discussed in §3.2.1. This surprising result suggests that earlier work, which confines edit operations to execution traces, would possibly be unable to repair the NETGEAR bugs. Testing this possibility more definitively would require developer-written regression tests, however. While fault localization is reasonable for source-level repairs (e.g., by definition the defect must be addressable in a visited statement), at the binary level repair can often be created by changing data [10, 32]. While fault localization does reduce the search time, for binary-level repairs it may cause a larger number of viable repair candidates to be excluded from the search space.

[4]https://github.com/eschulte/netgear-repair/blob/master/bin/test-cgi

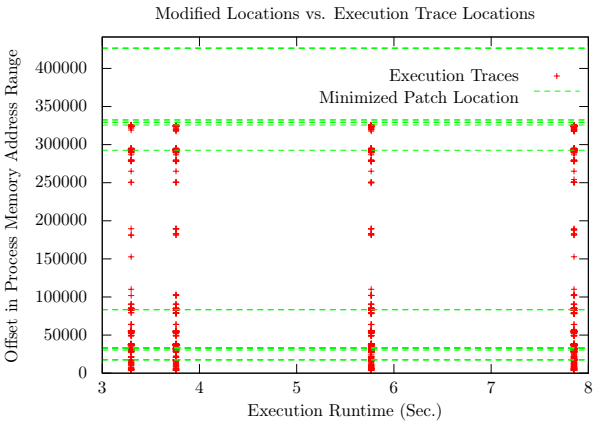

Figure 6: Code modifications occur in different locations from execution traces: The location of every edit in a minimized successful repair is plotted as a horizontal line. Only 2 of the 22 minimized edit locations are within 3 bytes of a sample from any test suite execution. Each vertical column shows points of execution traces from one test suite. Test suites shown from left to right are 3 tests (exploit tests only), 4, 7, and 11 tests (all exploit and author-generated regression tests), with 330, 399, 518, and 596 sampled execution locations respectively.

4.2.3 *The impact of Minimization*

In some cases the initial suggested repair, known as the *primary* repair, was not satisfactory. For example, suggested repairs sometimes worked when `net-cgi` was called directly on the command line but not through the embedded μHTTPd webserver,[5] or the repaired file failed to serve pages not used in the exploit test. However, Table 1 shows that in most cases the minimized version of the repair was satisfactory, successfully passing all hand-written regression tests, even those not used during the repair process.

As shown in Table 1, the initial evolved repair differed from the original at over 200 locations on average in the ELF program data, while the minimized repairs differed at only 1–3 locations on average. This great discrepancy is due to the accumulation of candidate edits in non-tested portions of the program data. Since these portions of the program were not tested, there was no evolutionary pressure to purge the harmful edits. Delta Debugging eliminates these edits.

Given the small number of edits which remain after minimization it is possible that, at least for those repairs which minimize to two diff windows (Column "Min Diff" in Table 1) corresponding to the two vulnerabilities in the original program, the use of a systematic exhaustive search that *only* retains program edits that strictly improve fitness may be sufficient to perform program repair.

4.2.4 *Iterative Repair*

The NETGEAR repairs required two distinct modifications, addressing two different vulnerabilities in a single evolutionary run. This is an instance of "iterative repair" of multiple vulnerabilities which has not previously been demonstrated in real-world software.

[5]http://wiki.openwrt.org/doc/uci/uhttpd

Run	Fit Evals	Full Diff	Min Diff	Full Fit	Min Fit
0	90405	500	2	8	22
1	17231	134	3	22	22
2	26879	205	2	21	22
3	23764	199	2	19	22
4	47906	319	2	6	6
5	13102	95	2	16	22
6	76960	556	3	17	22
7	11831	79	3	20	22
8	2846	10	1	14	14
9	25600	182	2	21	22
mean	33652.4	227.9	2.2	16.4	19.6

Table 1: The evolved repair before and after minimization. In these columns "Full" refers to evolved solutions before minimization and "Min" refers to solutions after. Columns labeled "Diff" report the number of unified diff windows against the original program data. The columns labeled "Fit" report fitness as measured with a full regression test suite, including the exploit tests. The maximum possible fitness score is 22, indicating a successful repair.

5. RELATED WORK

Evolutionary computation (EC) refers to the use of natural selection as a search heuristic [18, 21]. EC techniques have been developed to operate directly on machine code [22], and more recently they have been applied to the problem of software source-code repair [24, 1], optimization [39, 37, 23], to repairing assembly code and binary ELF files [34, 36], to combine different versions of software [13, 33], and to the incorporation of new functionality into existing software [16]. Software is inherently robust to mutation, a property termed *software mutational robustness* [38]. Software mutational robustness has been measured in software represented using source code, intermediate representations (e.g., Cil, LLVM IR), assembly code, and binary executables [35].

In addition to the EC methods mentioned above, Clearview [32] automatically patches errors in running binaries by learning invariants of running executables, and then reacting to attacks or bugs that invalidate the invariants by applying predefined patches.

6. DISCUSSION

The results presented here open up the possibility that end users could repair software vulnerabilities in closed source software without special information or aid from the software vendor.

6.1 Threats to Validity

There are several caveats associated with this initial work. First, we demonstrated repair on a single executable, and it is possible that our success in the absence of regression test suite will not generalize. However, our results do not appear to be based on any property unique to the NETGEAR vulnerabilities. We conjecture that our success at finding functional repairs is due to the beneficial impact of minimization and to *software mutational robustness* [38, 35].

We demonstrated our repairs running in a virtualized environment and not natively in the router. Although we did not test our repairs on physical NETGEAR WNDR3700 hardware, we are confident that our repairs would have the same

effect on hardware as they do in emulation, given that they affect aspects of program logic not directly related to the execution environment.

6.2 Future Work

Although security vulnerabilities are serious, an important implication of this technique is the ability of end users to change non-security aspects of software, i.e. the *customization* of COTS binaries by end users. This approach could be applied to *any* feature of program behavior which may be encoded in a fitness function. Although the direct synthesis of novel program behavior is certainly a more challenging task than the patching of vulnerabilities, there are many plausible and desirable yet simple software customizations, even including the direct removal of unwanted software features.

This technique could be combined with an automated testing or exploit generation technique, such as fuzz testing [29], to automatically "harden" closed source applications. Such a process would allow end users to proactively protect their devices from a wide range of easily generated attacks and could potentially be used to disrupt the large mono-culture of commercial firmware [19]. The technique could also be adapted by users to disable or break undesirable or insecure functionality (e.g., password reset) in closed-source applications. Finally, the technique could be distributed across multiple untrusted peers and used to distribute self-certifying patches [8, 36].

Whenever a patch is distributed there is the risk that a malicious individual will reverse-engineering an exploit from the patch text [5]. As shown in Table 1 our technique can generate edits that are not directly relevant to the repaired exploit. It may be possible to leverage these harmless, i.e. *neutral*, edits to reduce the risk of reverse engineering. By skipping the post-evolutionary minimization step, these irrelevant edits would be retained resulting in obfuscated patches which hide the relevant edits. However, skipping minimization would likely require a regression test suite or other method of ensuring semantic correctness.

7. CONCLUSION

The paper described a method that enables end users to repair COTS software without cooperation from the software vendor. This is accomplished through a number of novel extension to existing techniques of evolutionary program repair, including the use of user interaction and removing the requirements of a regression test suite and fault localization. We demonstrate the method by repairing two security vulnerabilities in the popular NETGEAR WNDR3700 router, vulnerabilities that currently exist in many actively used devices and to the authors knowledge have not been addressed by NETGEAR.

8. ACKNOWLEDGMENTS

We thank Z. Cutlip, who analyzed and announced the NETGEAR vulnerabilities and helped us reproduce the vulnerabilities locally; we thank M. Harmon, for discussions of automated program repair without a regression test suite; and S. Harding for suggesting the interactive regression repair algorithm. Partial support of this work provided by NSF (SHF-0905236), DARPA (FA8750-15-C-0118), AFRL (FA8750-15-2-0075), DHS (HSHQDC-14-C-00055), and the Santa Fe Institute. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the supporting agencies.

9. REFERENCES

[1] Andrea Arcuri. Evolutionary repair of faulty software. *Applied Soft Computing*, 11(4):3494–3514, 2011.

[2] Ashish Arora, Anand Nandkumar, and Rahul Telang. Does information security attack frequency increase with vulnerability disclosure? an empirical analysis. *Information Systems Frontiers*, 8(5):350–362, 2006.

[3] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[5] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 143–157. IEEE, 2008.

[6] Jonathan B Buckheit and David L Donoho. *Wavelab and reproducible research*. Springer, 1995.

[7] TIS Committee et al. Tool interface standard (tis) executable and linking format (elf) specification version 1.2. *TIS Committee*, 1995.

[8] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worm epidemics. *ACM Transactions on Computer Systems (TOCS)*, 26(4):9, 2008.

[9] Zachary Cutlip. Complete, persistent compromise of netgear wireless routers, October 2013. http://shadow-file.blogspot.com/2013/10/complete-persistent-compromise-of.html.

[10] Brian Demsky and Martin C.Rinard. Goal-directed reasoning for specification-based data structure repair. *Transactions on Software Engineering*, 32(12):931–951, 2006.

[11] Dennis Fisher. D-link planning to patch router backdoor bug, October 2013. http://threatpost.com/d-link-planning-to-patch-router-backdoor-bug/102581.

[12] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954. ACM, 2009.

[13] Blair Foster and Anil Somayaji. Object-level recombination of commodity applications. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 957–964. ACM, 2010.

[14] Stefan Frei, Bernhard Tellenbach, and Bernhard Plattner. 0-day patch exposing vendors (in) security performance. *BlackHat Europe, Amsterdam, NL*, 2008.

[15] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repairs: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering, 2012. ICSE 2012.* IEEE, 2011.

[16] Mark Harman, Yue Jia, and William B Langdon. Babel pidgin: Sbse can grow and graft entirely new functionality into a real world system. In *Search-Based Software Engineering*, pages 247–252. Springer, 2014.

[17] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill. Mips: A microprocessor architecture. *ACM SIGMICRO Newsletter*, 13(4):17–22, 1982.

[18] John Henry Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. The MIT press, 1992.

[19] IEEE security and privacy, special issue on IT monocultures. Vol. 7, No. 1, Jan./Feb. 2009.

[20] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[21] John R. Koza. Genetic programming: On the programming of computers by means of natural selection, 1992. *See http://miriad. Iip6. fr/microbes Modeling Adaptive Multi-Agent Systems Inspired by Developmental Biology*, 229, 1992.

[22] F. Kühling, K. Wolff, and P. Nordin. A brute-force approac to automatic induction of machine code on cisc architectures. *Genetic Programming*, pages 288–297, 2002.

[23] William B. Langdon and Mark Harman. Optimizing existing software with genetic programming. *IEEE Trans. on Evolutionary Computation*, 19(1):118–135, 2015.

[24] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automated software repair. *Transactions on Software Engineering*, 38(1):54–72, 2012.

[25] Claire Le Goues, Westley Weimer, and Stephanie Forrest. Representations and operators for improving evolutionary software repair. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 959–966. ACM, 2012.

[26] P Lougher and R Lougher. Squashfs-a squashed read-only filesystem for linux, 2006.

[27] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[28] Jill P Mesirov. Accessible reproducible research. *Science*, 327(5964):415–416, 2010.

[29] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[30] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.

[31] Peter Nordin, Wolfgang Banzhaf, and Frank D Francone. 12 efficient evolution of machine code for cisc architectures using instruction blocks and homologous crossover. *Advances in genetic programming*, 3:275, 1999.

[32] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102. ACM, 2009.

[33] Justyna Petke, Mark Harman, William B Langdon, and Westley Weimer. Using genetic improvement and code transplants to specialise a c++ program to a problem class. In *Genetic Programming*, pages 137–149. Springer, 2014.

[34] E. Schulte, S. Forrest, and W. Weimer. Automated program repair through the evolution of assembly code. In *25nd IEEE/ACM International Conference on Automated Software Engineering*, pages 313–16, 2010.

[35] Eric Schulte. *Neutral Networks of Real-World Programs and their Application to Automated Software Evolution*. PhD thesis, University of New Mexico, Albuquerque, USA, July 2014. `https://cs.unm.edu/ēschulte/dissertation`.

[36] Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. In *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2013.

[37] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *Proceedings of the nineteenth international conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2014.

[38] Eric Schulte, ZacharyP. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, pages 1–32, 2013.

[39] Pitchaya Sitthi-Amorn, Nicholas Modly, Westly Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics (TOG)*, 30(6):152, 2011.

[40] Westley Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.

[41] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Foundations of Software Engineering*, pages 253–267, 1999.

[42] Earl Zmijewski. Reckless driving on the internet, February 2009. http://www.renesys.com/2009/02/the-flap-heard-around-the-world/.