Software Engineering Institute

Well There's Your Problem: Isolating the Crash-Inducing Bits in a Fuzzed File

Allen D. Householder

October 2012

TECHNICAL NOTE
CMU/SEI-2012-TN-012

CERT Program

http://www.sei.cmu.edu

Carnegie Mellon

This material is based upon work funded and supported by the United States Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the

SEI Administrative Agent
AFLCMC/PZE
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

Table of Contents

List of Figures

Abstract

Mutational input testing (fuzzing, and in particular dumb fuzzing) is an effective technique for discovering vulnerabilities in software. However, many of the bitwise changes in fuzzed input files are not relevant to the actual software crashes found.

This report describes an algorithm that efficiently reverts bits from the fuzzed file to those found in the original seed file, keeping only the minimal bits required to recreate the crash under investigation. This technique reduces the complexity of analyzing a crashing test case by eliminating the changes to the seed file that are not essential to the crash being evaluated.

1 Introduction

Dumb fuzzing begins with an original, non-crashing seed file and produces mutated (i.e., fuzzed) files that cause crashes. A fuzzed file may cause different code paths to be executed than the seed file, but many of these differences are irrelevant to the eventual crash the fuzzed file induces in the software. Therefore, to facilitate crash analysis, we need to reduce the differences between the seed file and the fuzzed file while still retaining a test case that can induce the crash that was initially detected. In this report, we describe an algorithm that efficiently reverts bits from the fuzzed file to those found in the original seed file, keeping only the minimal bits required to recreate the crash under investigation.

1.1 Determining Crash Uniqueness

The technique described here depends on the ability to identify unique crashes. We will briefly explain how this technique can be used for doing so, but the results of the paper are not dependent on the specific method of crash-uniqueness determination.

It is not sufficient to merely detect that a candidate file causes a crash, as there may be many possible crashes caused by a mix of input data. Thus we use a fuzzy hashing technique based on dynamic test generation for uniquely identifying crashes on Linux systems using the Gnu Debugger (GDB) [4]. For each crashing fuzzed file found, we gather GDB output and parse the last few lines of backtrace to determine the crash characteristics. If the backtrace contains a string that identifies a specific line in the source code (e.g., at *foo.c:37*) indicating that debugging symbols are present, we use the source line information. Otherwise, we take the address of each backtrace line. We currently use five lines of backtrace for the uniqueness determination.

1.2 Prior Work

This work was inspired by Dan Rosenberg's *FuzzDiff* tool, which takes a heuristics-based approach to the problem of test case minimization [5].

2 Rationale

Consider the problem space: there are two files, a *SeedFile* that does not cause a crash, and a *FuzzedFile* that induces a crash. The *FuzzedFile* was derived from the *SeedFile* by probabilistically flipping bits.[1] Because both files are the same size, we can use the Hamming Distance *HD()* as the metric we seek to minimize.

We do not know how many bits are minimally needed to induce the crash, but we know that it must be in the range $1 \leq min \leq HD(SeedFile, FuzzedFile)$.

When attempting to iteratively minimize the Hamming Distance between the *SeedFile* and the *FuzzedFile*, there are two key questions to answer:

1. How many bits should we attempt to revert back to the *SeedFile* and the *FuzzedFile* value in this iteration?
2. Have we reached a minimum?

We tackle these questions in Sections 2.1 and 2.2.

2.1 Optimizing the Number of Bits to Revert

Deciding how many bits to revert in any given iteration is critical to finding an optimal solution. Calculate the probability of success in a given iteration as well as the reduction in the Hamming Distance (i.e., the number of bits restored from the *SeedFile*) when the iteration is successful. Multiply these two values to determine an expected reduction for each possible set of parameters. Then simply select the parameters with the maximum expected reduction.

2.1.1 Expected Hamming Distance Reduction

Calculating the reduction in Hamming Distance is straightforward. Simply iterate through the *FuzzedFile* and restore a bit from the *SeedFile* with the probability p to produce a *NewFile* such that

$$NewHD = HD(SeedFile, NewFile) \quad (1)$$

and

$$0 < NewHD < HD\ (SeedFile, FuzzedFile) \quad (2)$$

Thus the reduction in distance will be

$$BitReduction = p \times HD\ (SeedFile, FuzzedFile) \quad (3)$$

Note that this is equivalent to saying that

$$NewHD = (1 - p) \times HD\ (SeedFile, FuzzedFile) \quad (4)$$

2.1.2 Probability of Success in a Given Iteration

Once we know the potential *NewHD*, we can recast the problem as one of drawing marbles from an urn and apply the hypergeometric distribution. Consider the following example taken from

[1] In our fuzzing environment, we currently use zzuf [1] but any randomized file fuzzing technique would work.

E.T. Jaynes' *Probability Theory* [2]: An urn contains N marbles, M of which are red, the remainder white. If we draw n marbles from the jar, the probability of drawing r red marbles is

$$P(r|N,M,n) = \frac{\binom{M}{r}\binom{N-M}{n-r}}{\binom{N}{n}} \tag{5}$$

In this case, we let $N = HD$ (*SeedFile, FuzzedFile*), M = the minimum number of bits to induce the same crash (unknown), and $n = NewHD$ from equation (4). Furthermore, for our trial to be a success, we would have to draw all the red marbles at once. So we can set $r = M$. Thus equation (5) becomes

$$P(M|N,M,n) = \frac{\binom{N-M}{n-M}}{\binom{N}{n}} \tag{6}$$

Which further reduces[2] to

$$P(M|N,M,n) = \frac{n!\,(N-M)!}{N!\,(n-M)!} \tag{7}$$

We still do not know what M is. However, we can begin by hypothesizing that $M = 1$ and then refine our hypothesis as we go (see Section 2.2).

Given that we know

$$N = HD(SeedFile, FuzzedFile) \tag{8}$$

and we hypothesize $M = 1$, using equation (4) we can calculate

$$n_i = (1 - p_i)N \tag{9}$$

and thus $P(M|N, M, n_i)$ for each p_i where

$$p_i = i/N \tag{10}$$

and i is an integer in the range $1 \leq i < N$.

We can then multiply the resulting $P(M|N, M, n_i)$ values by the

$$BitReduction = (N - n_i) \tag{11}$$

and choose a

$$DiscChance = p_{max} \tag{12}$$

that satisfies

$$p_{max} = p : max(P(M|N, M, ni) \times (N - ni)) \tag{13}$$

with $1 \leq i < N$.

---

[2] N, M, and n can be in the thousands or even millions at the start of a minimization. Thus, when implementing this algorithm in code, we found it convenient to transform equation (7) using the log gamma function $\ln\Gamma()$ in lieu of factorials to avoid extremely large integers.

With this series of calculations we have now determined an answer to Question 1 in Section 2: How many bits should we attempt to revert back to the *SeedFile* value in this iteration? The answer is *DiscChance* × *HD*(*SeedFile, FuzzedFile*) bits.

Pseudocode for this algorithm is in Algorithm 1.

Algorithm 1: Setting the Discard Chance

```
for 1 ≤ i ≤CurrHD do
    CDChance[i] ← i/CurrHD
end for
for all CDChance[i] do
    Calculate P_hit [i] [see (7)]
    BitReduction[i] ← CurrHD × CDChance[i]
    ExpectedReduction[i] ← P_hit [i] ×
    BitReduction[i]
end for
DiscChance ← max(ExpectedReductions)
return DiscChance
```

2.2 Iterative Hypothesis Testing

Next we address Section 3, Question 2: How do we know whether we have reached a minimum? There are three ways to determine whether we have reached a minimum:

1. We find a *NewFile* that induces the same crash as the *FuzzedFile* with *NewHD* = 1.
2. We have exhaustively searched all files with a Hamming Distance smaller than *NewHD* and found that none of them induces the same crash as the *FuzzedFile*.
3. We have tried at least *x* times and failed to find a file with a Hamming Distance smaller than *NewHD* that induces the same crash as the *FuzzedFile*. We define this as a *hit*.

Of these three ways, both one and two are just convenient shortcuts. The "meat" of the problem lies in deciding what *x* should be in three.

Our approach is to iteratively test the hypothesis until there is at least one hit left to be found in the search space. Here we use the identity that

$$p_{hit} = 1 - p_{miss} \tag{14}$$

and observe that the chance of getting at least one hit in *x* tries is

$$P(\geq 1\_hit\_in\_x\_tries) = 1 - p^{x}_{miss} = 1 - (1 - p_{hit})^{x} \tag{15}$$

Another way to interpret equation (15) is that if we try *x* times and failed to find a hit, then P (≥ 1 *hit_in_x_tries*) is equal to our confidence that p_{hit} is incorrect. So we can choose our desired confidence value C (e.g., $C = 0.999$) such that

$$C = 1 - (1 - p_{hit})^{x} \tag{16}$$

and then solve for *x*

$$x = \frac{\ln(1 - C)}{\ln(1 - p_{hit})} \tag{17}$$

where $p_{hit} = P(M \mid N, M, n)$ from equation (7).

If we reach a point where we have x consecutive misses with a given set of parameters, we have confidence C that p_{hit} must be wrong. For p_{hit} to be wrong, one of its input variables must be wrong. However, in equation (7) N is measured, and n was chosen based on the calculations in Section 2.1, which in turn are based on M . So this leaves M as the variable we should update.

Recall from Section 2.1.2 that we started by hypothesizing that the target size is $M = 1$. We can then conclude that given x misses, $M > 1$. Therefore, when these conditions are met, we increment our target size guess $M' = M + 1$, then repeat the calculations of Section 2.1 to find a new *Disc-Chance* and iterate.

When we reach a state where $M = NewHD$ (i.e., we conclude with confidence C that the target is at least M bits, and we have found a *NewFile* at that Hamming Distance), we end the search and declare *NewFile* to be the minimized version of the *FuzzedFile*.

Pseudocode for the minimization algorithm is shown in Algorithm 2.

Algorithm 2: Main Minimization Algorithm

```
MinDistFound ← HD(SeedFile, FuzzedFile)
CurrentFile ← FuzzedFile
MissCount ← 0
TargSzGuess ← 1
MinFound ← False
while MinFound ≠ True do
   DiscChance ← GetDiscChance() (see 1) AllowedMisses ← GetAllowedMisses() [see (17)]
   while MissCount < AllowedMisses do
      Generate a NewFile by replacing bits in the FuzzedFile with bits from the SeedFile with probability Disc-
      Chance
      if CrashID(NewFile) =
      CrashID(FuzzedFile) then
         MinDistFound
         HD(SeedFile, NewFile)
         CurrentFile ← NewFile
         reset DiscChance and AllowedMisses based on the new MinDistFound
         MissCount ← 0
         if MinDistFound = TargSzGuess then
            MinFound ← True
         end if
         break inner loop
      else
         increment MissCount
    end if
    end while
   if MinFound ≠ True then
      TargSzGuess ← TargSzGuess +1
   end if
end while
```

3 Results and Discussion

We have been using this minimization technique in fuzzing environments for some time now, and it has helped us distill crashing test cases to their minimum difference from a non-crashing test case.

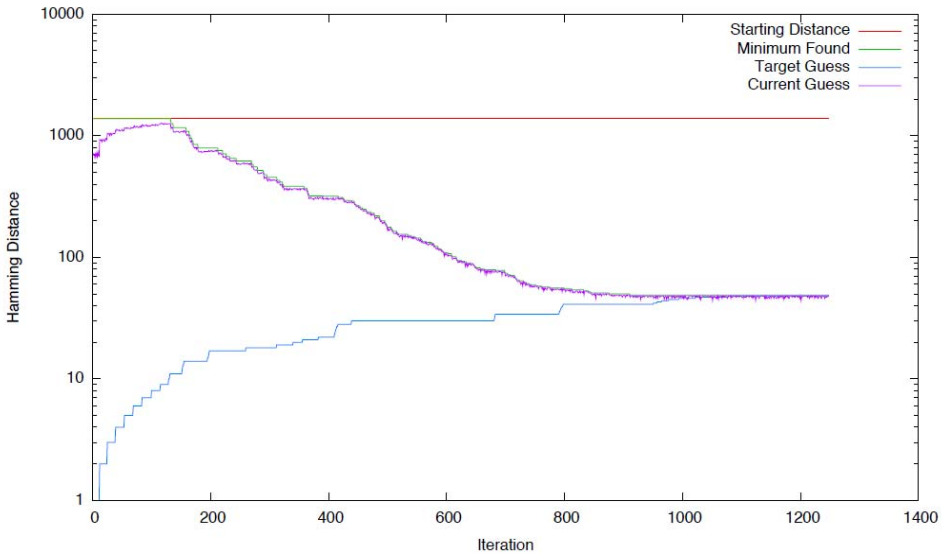

Figure 1 shows the performance of the minimization algorithm on a crashing test case that started with a Hamming Distance of 1383 bits. Using the confidence level $C = 0.999$, it took 1248 tries to minimize the crash to a Hamming Distance of 49 bits. In this instance, the algorithm found its first hit after 133 attempts with a target size guess of 11 bits.

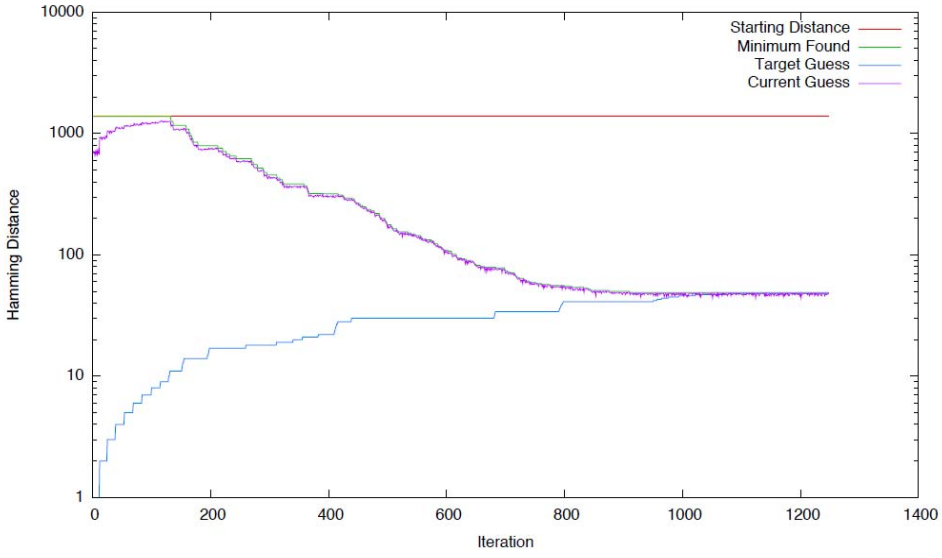

Figure 1: Test Case Minimization Example

From that point on, the algorithm continued to find smaller and smaller minima following an approximately exponential curve up to the point where the target size guess and minimum found are nearly equal (around iteration 800). From there, the minimization progress slows as the algorithm completes its convergence.

To evaluate the performance of the minimization algorithm, we compared the number of iterations required for the algorithm to terminate to the final Hamming Distance of the minimized test case. Data was taken from a series of fuzzing campaigns against multiple applications and with multiple seed files.

As shown in

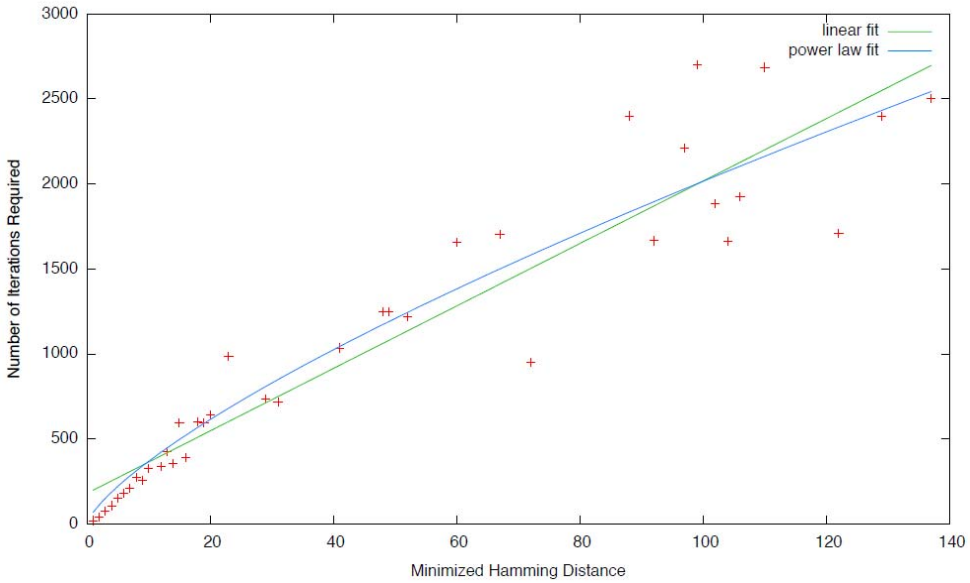

Figure 2, general algorithm performance appears to be approximately linear relative to the Hamming Distance of the minimized test case. When multiple test cases minimized to the same Hamming Distance, Figure 2 shows the average number of iterations for that distance.

The linear fit is

$$y = 18.4x + 179.7 \tag{18}$$

with $R^2 = 0.89$.

The power law fit is

$$y = 32.6x^{0.92} \tag{19}$$

with $R^2 = 0.96$.

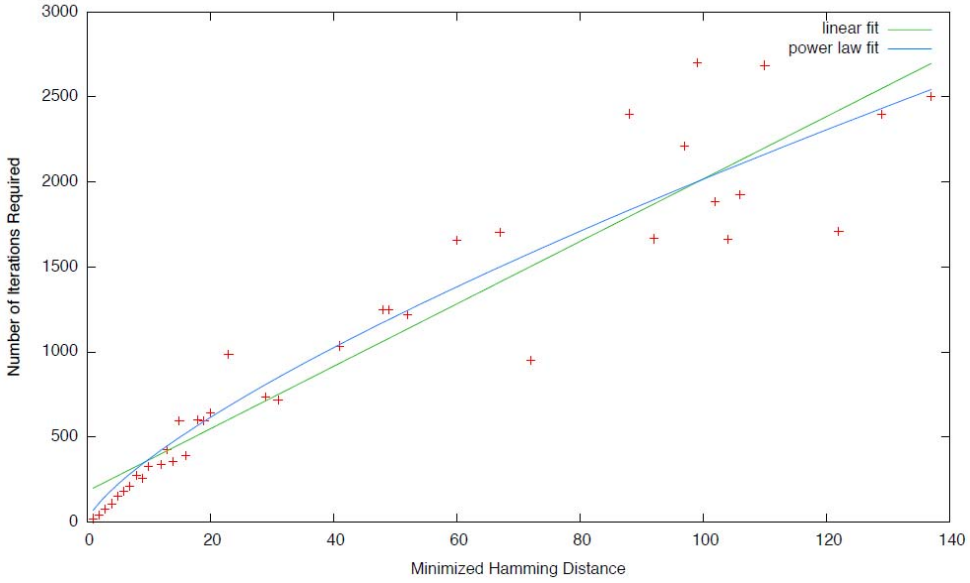

Figure 2: Algorithm Performance

3.1 Limitations and Future Work

The minimization algorithm presented here assumes that both the original seed file and the fuzzed test case to be minimized are of the same length, thereby allowing the use of the Hamming Distance as the metric to be minimized. We believe it should be possible to generalize the algorithm to use Levenshtein Edit Distance to enable the application of this method to fuzzing techniques that alter the input file length.

Furthermore, it is possible that the same software bug could cause different crash signatures using the uniqueness determination described in Section 1.1. In particular, applications that implement Just-In-Time (JIT) compilation tend to resist test case minimization since varying the input can alter the JIT code that is executed, thus making it harder to recreate the same crash in the minimization process.

Although it is rare, we have seen in some cases that the exact same test case file can cause different crash signatures to occur; so as a practical workaround, we conduct a quick check before we begin the main minimization loop to determine if the crash signature is stable with multiple runs of the test case.

Finally, the algorithm is, of course, at the mercy of the fuzzed files to which it is applied. Although it is both effective and efficient at finding the minimal changes required to recreate a crash given a particular fuzzed file, it is entirely possible that another fuzzed file for the same crash could in fact minimize to a smaller Hamming Distance. In practice, this difference has not been an issue, since most crashing test cases minimize down to a relatively small Hamming Distance (around 5-20 bits is typical).

4 Conclusion

Fuzzing in general—and dumb fuzzing in particular—has been shown to be an effective technique for discovering vulnerabilities in software. However, many of the bitwise changes in fuzzed input files are not relevant to the actual software crashes found.

In this report, we described an algorithm that efficiently reverts those non-essential bits from the fuzzed file to those found in the original seed file. This algorithm reduces the complexity of analyzing a crashing test case by eliminating bitwise changes that are not essential to the crash being evaluated.

We have found this technique to be very useful to our vulnerability analysts in their ongoing efforts to discover, analyze, and coordinate vulnerabilities with software vendors.

An implementation of the minimization algorithm described here is available in the CERT Basic Fuzzing Framework (BFF) for Linux and Mac OS X, and the CERT Failure Observation Engine (FOE) for Windows, which are released under open source licenses. CERT BFF can be downloaded at http://www.cert.org/vuls/discovery/bff.html while CERT FOE is available from http://www.cert.org/vuls/discovery/foe.html.

Bibliography

[1] S. Hocevar. (2011, May 12). *zzuf—multi-purpose fuzzer*. [Online]. Available: http://caca.zoy.org/wiki/zzuf

[2] E. T. Jaynes and G. L. Bretthorst, *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.

[3] C. Miller. (2007, August 3). *How Smart Is Intelligent Fuzzing-or-How Stupid Is Dumb Fuzzing*. [Online]. Available: http://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-miller.pdf

[4] David Molnar, Xue Cong Li, and David A. Wagner, "Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs," in *Proceedings of the 18th conference on USENIX security symposium, SSYM'09*, Berkeley, CA, 2009, pages 67–82.

[5] Dan Rosenberg. (2010) *FuzzDiff*. [Online]. Available: http://code.google.com/p/fuzzdiff/

[6] A. Takanen, J. DeMott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*. Artech House Publishers, 2008.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	October 2012	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Well There's Your Problem: Isolating the Crash-Inducing Bits in a Fuzzed File	FA8721-05-C-0003

6. AUTHOR(S)

Allen D. Householder

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2012-TN-012

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116	n/a

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Mutational input testing (fuzzing, and in particular dumb fuzzing) is an effective technique for discovering vulnerabilities in software. However, many of the bitwise changes in fuzzed input files are not relevant to the actual software crashes found.

In this report, we describe an algorithm that efficiently reverts bits from the fuzzed file to those found in the original seed file, keeping only the minimal bits required to recreate the crash under investigation. This technique reduces the complexity of analyzing a crashing test case by eliminating the changes to the seed file that are not essential to the crash being evaluated.

14. SUBJECT TERMS	15. NUMBER OF PAGES
fuzzing, fuzz testing, automated debugging, software testing, adaptive testing	19

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL