# Protecting your software updates

*Bart Coppens*
*bart.coppens@elis.ugent.be*
*Ghent University*
*Sint-Pietersnieuwstraat 41*
*B-9000 Gent*
*Belgium*
*Telephone: +32 9 264 98 54*
*Fax: +32 9 264 35 94*

*Bjorn De Sutter*
*bjorn.desutter@elis.ugent.be*
*Ghent University*
*Sint-Pietersnieuwstraat 41*
*B-9000 Gent*
*Belgium*

*Koen De Bosschere*
*koen.debosschere@elis.ugent.be*
*Ghent University*
*Sint-Pietersnieuwstraat 41*
*B-9000 Gent*
*Belgium*

**Abstract**

As described in many blog posts and scientific literature, exploits for software vulnerabilities are often engineered on the basis of patches. This involves the manual or automated identification of the vulnerable code. We evaluate how this identification can be automated with the most frequently referenced diffing tools. We demonstrate that for certain types of patches, but not for all, those tools are indeed effective attacker tools. We also demonstrate that by using binary code diversification, the effectiveness of the tools can be diminished severely, thus severely closing the attacker's window of opportunity.

**Keywords:** software protection, binary code diversity, diffing tools, patch-based attacks

## Patch-based exploit development

Every second Tuesday of the month, Microsoft releases its *Patch Tuesday* software updates. These updates include security patches, most of which are documented to inform system administrators what they are vulnerable to. Microsoft typically words this without giving concrete hints on how to exploit the fixed vulnerabilities. But their descriptions do not always match the patched vulnerabilities, and some are not even mentioned [5].

So when security researchers or attackers get their hands on the binary patches, they start inspecting them in preparation of *Exploit Wednesday*, the attacker's window of opportunity to target unsuspicious users that did not immediately apply the update. In some cases, attackers can also target users that did apply the patch immediately, but were left vulnerable because a fix was not complete [7]. With the help of so-called diffing tools like Darungrim (`http://www.darungrim.org/`) and BinDiff (`http://www.zynamics.com/bindiff.html`), the attackers compare the binary code before and after the patch to identify the fixed code fragments and the applied fixes, to determine the closed vulnerabilities, and ultimately to devise actual exploits for those vulnerabilities. While Patch Tuesday and Exploit Wednesday are the best know instances of this scenario, the same scenario takes place every time software vendors release binary security updates.

Similar attacks can be used to identify the code that implements new functionality in software updates, as a first step towards the reverse-engineering or theft of its intellectual property. Such attacks and diffing tools can also be used to port information from one program version to another. When attackers can identify the corresponding parts in consecutive program versions, they can more easily reuse their existing reverse-engineering knowledge. Barthen relied on a diffing tool to transfer debugging information obtained from an older version of World of Warcraft to a newer version that was distributed without debugging information [2].

Brumley et al demonstrated that sometimes exploits can be devised without any human code analysis or understanding. All their attack needs is an accurate identification of the new instructions inserted by a patch [3]. On that basis, they apply constraint solving techniques on program inputs to generate attacks automatically. To the best of our knowledge, all published patched-based attacks [3, 8, 9, 12, 13, 14, 15] similarly build on the assumption that expert attackers assisted by diffing tools can easily identify the relatively few relevant differences between unpatched and patched binaries.

In this article, we demonstrate that this assumption is often but not always valid for ordinary code. More importantly, we demonstrate that it falls completely apart for diversified code. In this context, diversification is the deliberate generation of different binaries from the same source code. We evaluate how diversification complicates and delays the identification of the vulnerable code by making the diffing tools less effective. A delay shortens the attacker's window of opportunity, thus reducing his potential profit. It also corresponds to an increased effort to compensate the reduced effectiveness of the tools, and hence to an increased investment for the attacker. As such, diversification primarily targets economic goals, rather than full technical protection, which is impossible anyway.

## Modeling Attack Effort and Delay

To evaluate the attack effort and the incurred delay, we model it in terms of quantitative program properties that relate to real attacks.

When patches change the semantics of a program at specific source code locations, the compiled binary will incur semantic mutations at corresponding locations. In addition, many (small) syntactic mutations will be spread through the binary as a result of changed code offsets and global compiler optimizations. Security researchers and attackers use diffing tools to differentiate the syntactic mutations from the semantic ones. The tools compare abstract representations of the original and patched binaries and return a score for each code fragment indicating where completely or partially equivalent counterparts were found in the other binary. This matching score can be graphical as in Figure 1 or numerical. Using his experience, the matching scores and many heuristics, the attacker then implicitly or explicitly ranks all fragments and inspects them manually or automatically to identify the semantic mutations that reveal vulnerabilities. Attackers can use the same approach to identify, reverse-engineer, and lift new functionality from software updates.

To model the attacker's additional effort incurred by diversification, we developed scripts that automate the application of commonly-used diffing tools and heuristics. When we apply the scripts on a

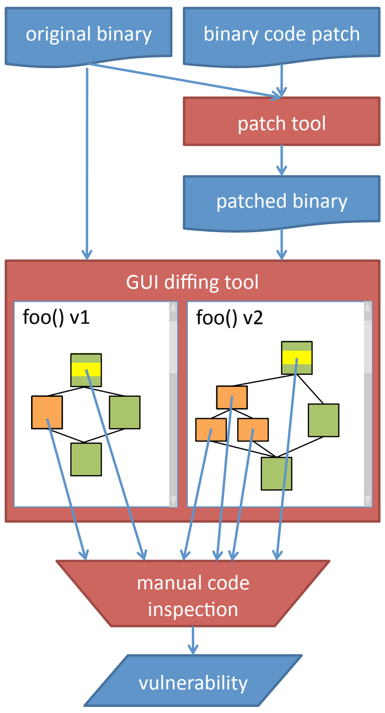


Fig. 1: Patch-based attack flow

patch of which we know the ground truth, we can compute the ranking of all relevant instructions. This ranking corresponds to the number of instructions that the attacker will have to inspect and analyze manually or that his tools will have to target automatically. The additional effort incurred by an attacker because of diversification can therefore be modeled by the difference in the relevant instructions' ranking with and without diversification.

How that additional effort delays an attacker depends on his approach. When an attack continues with the manual inspection of the supposedly most interesting code fragments, the additional effort incurred by diversification can only be parallelized at very high cost. Therefore a delay proportional to the decreased effectiveness of the diffing tools is to be expected.

For a specific type of security patches, namely newly inserted input-validity checks, the automated patch-based exploitation generation (APEG) approach by Brumley et al can be parallelized almost completely by running the technique in parallel for every potential new check identified by the diffing tools [3]. This seems to imply that for this type of security patch, attackers with a large amount of computing resources in the cloud or in botnets, would not be delayed significantly by software diversification. However, APEG initially treats all conditional branches found in the patched program but not in the original program as potential new checks. For each of these branches it will try to construct program inputs that trigger the check in the patched program and that cause a security property to be violated in the original program. For conditional branches that do not correspond to input-validity checks but that merely reflect syntactic changes that thwarted the diffing tools, the iterative search for ever more complex inputs that violate security properties will explode combinatorially, making the APEG approach much less efficient, if useful at all, as acknowledged by the authors [3]. So even when exploits can in theory be generated automatically, a significant delay related to the drop in effectiveness of those tools can be expected.

## Case studies

Buffer overflows and integer overflows are among the most common program vulnerabilities. Causes include insufficient input-validity checking, coding errors such as neglecting the null character that terminates strings, and unsafe buffer manipulation APIs. The three security cases we evaluate are all patches for these common vulnerabilities. We also added a functionality patch that replaces one algorithm by

```
es = -1;
N = 1;
do {
    if (N >= 2*1024*1024) RETURN(BZ_DATA_ERROR);
    if (nextSym == BZ_RUNA) es = es + (0+1) * N; else
    if (nextSym == BZ_RUNB) es = es + (1+1) * N;
    N = N * 2; if (N >= 2*1024*1024) RETURN(BZ_DATA_ERROR);
```

(a) bzip2 patch

```
#define PNG_tIME_STRING_LENGTH ~~30~~ 29

png_strncpy(tIME_string,
                png_convert_to_rfc1123(read_ptr, mod_time),
                PNG_tIME_STRING_LENGTH);
tIME_string[PNG_tIME_STRING_LENGTH] = '\0';
```

(b) png_debian patch

```
#define PNG_tIME_STRING_LENGTH ~~30~~ 29

~~png_strncpy~~png_memcpy(tIME_string,
                png_convert_to_rfc1123(read_ptr, mod_time),
                PNG_tIME_STRING_LENGTH);
tIME_string[PNG_tIME_STRING_LENGTH] = '\0';
```

(c) png_beta patch

Fig. 2: Three of the four source code patches

another. This case represents scenarios of patch-based IP theft.

The first security patch we evaluate, hereafter called `bzip2`, fixed the integer overflow vulnerability CVE2010-0405 in the `bzip2` program by inserting a check as indicated in Figure 2(a). This patch is similar to input-validity checks to protect against buffer overflows. In the binary, it inserts a short instruction sequence as shown in Figure 3(a).

Our second security patch is an off-by-one fix for vulnerability CVE-2008-3964 in the `pngtest` utility. The fix decrements a hard-coded, incorrect buffer length as shown in Figure 2(b). We will refer to this patch, which was distributed by the Debian GNU/Linux distribution as a separate patch, as `png_debian`. In the binary it resulted in four mutations to immediate operands: in two similar fragments the constant 30 is replaced by 29 and the absolute address of `tIME_string[30]` is replaced by that of `tIME_string[29]`. Figure 3(b) shows one of those changed fragments.

In other distributions, this fix was part of a larger update from `libpng` 1.2.23-beta01 to beta02 that also contains a lot of patches not related to CVE-2008-3964. In that larger update, the relevant code fragments were patched as shown in Figure 2(c). In addition to the changed length, the call to `png_strncpy` is replaced by a call to `png_memcpy`. The compiler inlines that call and unrolls its data copying loop, so in the patched binary the call to `png_strncpy`, including the preparation of arguments, is replaced by a sequence of `mov` instructions as shown in Figure 3(c). We refer to this security patch as `png_beta`.

Finally, we have chosen the SPEC benchmark program `soplex` and a patch that replaces two (out of several more) calls to `quicksort` with calls to a newly added `mergesort`. We call this functionality patch `soplex`. With this SPEC benchmark program come training and reference inputs that enable us to conduct valid performance experiments.

We compiled and statically linked the original and patched source code on Linux with `gcc -O3`[1]. Table 1 shows the patched binary sizes as well as the sizes of the binary patches generated with the `bsdiff` tool. It are those patches that are typically distributed to end users. The three relatively large patch sizes indicate that those patches involve many syntactic mutations, which the attacker has to weed

[1] As a result of the static linking, the binary modules fed to the diffing tools are quite large. This is in line with the trend we observe on Microsoft Windows systems where applications are distributed more and more as few large dynamically linked libraries (DLLs) instead of many small ones to reduce the number of inter-DLL interfaces that expose sensitive symbolic information to attackers. As such, we believe our results can be extrapolated to, e.g., Windows systems.

```
0x080538b4:  movl  $0xffffffff,0x60(%esp)
0x080538bc:  movl  $0x1,0x5c(%esp)
0x080538c4:  cmpl  $0x1ffff,0x5c(%esp)
0x080538cc:  mov   $0xfffffffc,%esi
0x080538d1:  jg    0x8052892 <BZ2_decompress+386>
```

(a) bzip2 patch

```
0x0804924e:  mov   0x124(%esp),%eax
0x08049255:  mov   %eax,(%esp)
0x08049258:  call  0x804a8c0 <png_convert_to_rfc1123>
0x0804925d:  mov   %eax,0x4(%esp)
0x08049261:  movl  $~~0x1e~~0x1d,0x8(%esp)
0x08049269:  movl  $0x80d9010,(%esp)
0x08049270:  call  0x80806b0 <strncpy>
0x08049275:  incl  0x80d900c
0x0804927b:  movb  $0x0, ~~0x80d902e~~0x80d902d
0x08049282:  jmp   0x8048b44
```

(b) png_debian patch

```
0x804927e:   mov   0x124(%esp),%eax
0x8049285:   mov   %eax,(%esp)
0x8049288:   call  0x804a920 <png_convert_to_rfc1123>
~~0x804928d:   mov   %eax,0x4(%esp)~~
~~0x8049281:   movl  $0x1e,0x8(%esp)~~
~~0x8049289:   movl  $0x80d9010,(%esp)~~
~~0x8049290:   call  0x80806b0 <strncpy>~~
0x804928d:   mov   (%eax),%ebp
0x804928f:   mov   %ebp,0x80d9010
0x8049295:   mov   0x4(%eax),%edi
0x8049298:   mov   %edi,0x80d9014
0x804929e:   mov   0x8(%eax),%esi
0x80492a1:   mov   %esi,0x80d9018
0x80492a7:   mov   0xc(%eax),%ebx
0x80492aa:   mov   %ebx,0x80d901c
0x80492b0:   mov   0x10(%eax),%ecx
0x80492b3:   mov   %ecx,0x80d9020
0x80492b9:   mov   0x14(%eax),%ebp
0x80492bc:   mov   %ebp,0x80d9024
0x80492c2:   mov   0x18(%eax),%edi
0x80492c5:   mov   %edi,0x80d9028
0x80492cb:   movzwl 0x1c(%eax),%eax
0x80492cf:   incl  0x80d900c
0x80492d5:   mov   %ax,0x80d902c
0x80492db:   movb  $0x0,~~0x80d902e~~0x80d902d
0x80492e2:   jmp   0x8048b44
```

(c) png_beta patch

Fig. 3: Semantic changes in three of the four binary code patches

out.

## Diffing Tools and Heuristics

IDA Pro (`http://www.hex-rays.com/products/ida/index.shtml`) is by far the most used tool to disassemble and inspect binary code [6]. IDA Pro disassembles the binary code into assembler instructions, which are then partitioned into basic blocks (i.e., single-entry single-exit sequences of instructions) and functions, on the basis of which control flow graphs (CFGs) and a call graph are constructed. Users can then invoke plug-ins to analyze or transform the constructed graphs. Code disassembly, CFG construction and call graph construction are notoriously difficult tasks, in particular when programs have been obfuscated [4]. IDA Pro therefore occasionally disassembles and partitions code incorrectly. Consequently, the plug-ins have to operate on incomplete, incorrect CFGs, which affects their effectiveness.

Several diffing plug-ins exist that try to identify the matching fragments in two program versions. They use different, mostly undocumented algorithms, abstraction levels, heuristics, and scoring systems for matching fragments. We evaluated four of them on IDA Pro v5.7.0.935:

- PatchDiff2 (`http://code.google.com/p/patchdiff2/`)
- TurboDiff (`http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=turbodiff`),
- BinaryDiffer (`http://code.google.com/p/binarydiffer/`)
- BinDiff (`http://www.zynamics.com/bindiff.html`).

Of the four tools, only BinDiff and TurboDiff produced good results on our use cases, so we do not report results of the other tools.

TurboDiff first tries to match functions, in which it then tries to find matching basic block pairs. The tool's output differentiates between *completely identical*, *mutated*, and *unmatched* basic blocks. As IDA Pro does not disassemble all code, TurboDiff also leaves some blocks *not reported.* BinDiff first also matches functions, in which it then tries to match basic blocks, in which it then tries to match instructions. BinDiff reports *matched* and *unmatched* instructions, and some instructions remain *not reported.*

The diffing tools thus partition the code fragments into categories that can be ranked by the attacker and his heuristics. For example, code fragments that are categorized as not having changed at all are less likely to contain semantic mutations. There is no absolute guarantee, however. When pairs of fragments are reported as identical, this only means that the similarity as computed by the tool at its particular abstraction level is above some threshold, not that the fragments are really identical.

For that reason, we always report both the recall and the pruning factor obtained with a tool and heuristics. Recall is the fraction of all relevant instructions retrieved [11]. For an attacker tool to be effective, it has to retrieve as much as possible of the instructions that encode semantic mutations. So higher recall is better. The pruning factor relates to the standard metric of precision [11], which is the fraction of all retrieved instructions that are relevant. We do not report precision, however, because precision depends as much on the patch's size as it depends on the number of retrieved instructions. Precision is therefore not a good metric to model an attacker's effort or delay. Instead we report the pruning factor, which is the fraction of all instructions not retrieved, i.e., the fraction of the whole program pruned away for further inspection. As its pruning factor increases, a tool becomes more effective.

Besides ranking the different fragment categories as determined by a tool, attackers use heuristics and perform additional actions with the tools. We investigated the following potentially useful heuristics and optimizations:

1. *Code not triggered by inputs is less relevant.* The rationale for this assumption is that both tools and humans face the undecidable problems of determining which execution paths through a program are feasible and which code is reachable. To avoid wasting time on infeasible execution paths or unreachable code, attackers will focus on code of which they can demonstrate the reachability by executing the program on chosen inputs.

2. *Extend the disassembly process of IDA Pro.* When IDA Pro's automatic recursive-descent disassembly process [6] does not disassemble all code, this can result in fewer matches being reported

| benchmark | undiversified | | low diversification | | | high diversification | | |
|---|---|---|---|---|---|---|---|---|
| | patched binary size (kB) | bsdiff binary patch size (kB) | average patched binary size overhead | average binary patch size overhead | average patched binary performance overhead | average patched binary size overhead | average binary patch size overhead | average patched binary performance overhead |
| bzip2 | 401 | 4.23 | 20% | 4631% | 20% | 28% | 5084% | 34% |
| png_debian | 502 | 0.15 | 20% | 168946% | 39% | 29% | 184772% | 60% |
| png_beta | 502 | 8.91 | 20% | 2743% | 28% | 29% | 3040% | 41% |
| soplex | 834 | 10.93 | 20% | 4011% | 23% | 31% | 4482% | 48% |

Tab. 1: Original program sizes and overhead of using diversification

and hence more work left for the remainder of the attack. To avoid this, the attacker can use IDA Pro's interactive operation to indicate additional locations in the binary to be disassembled. For example, he can make sure that all code that got executed for his inputs gets disassembled. We implemented an automated plug-in that mimics this manual optimization to model the strongest possible attacker.

3. *Code not reported by the diffing plug-ins is irrelevant.* As an alternative to the extended disassembly, we also investigated what would happen if the attacker would simply consider all originally non-disassembled code irrelevant. This models a non-expert IDA Pro user.

4. *Filter certain patterns in the CFGs.* With this heuristic, we filter out clear cases of purely syntactic changes such as no-op instructions like `add zero` and chains of control flow transfers that can be replaced by single transfers. Moreover, we filter out unmatched basic blocks that IDA Pro assigned to some function but that are (according to IDA Pro) unreachable from the function's entry point. We observed that such basic blocks typically originate from autogenerated code such as stack unwinding code, in which case it will not include vulnerabilities anyway, or from IDA Pro's bad partitioning of code into functions. In the latter case, the rationale is that blocks being reported as unreachable in their function have more likely not been matched because of the bad partitioning of code into functions, than because there is no matching counterpart in the other binary.

5. *Expand the observed instruction window.* In practice, attackers will browse CFGs displayed side by side as in Figure 1. Without any effort they then also observe the instructions surrounding the instructions categorized by the tool as mutated. We can model this behavior by expanding the sequences of instructions that the tool has categorized as mutated. For this paper, we expand those instruction sequences with surrounding instructions that are mutated according to the ground truth. This models experienced attackers that can instantly prune all irrelevant code displayed as not mutated. This expansion also allows us to evaluate the extent to which the tools guide attackers towards the relevant instructions even when the tool don't retrieve them directly.

## Results

Figure 4 presents the results obtained with ten combinations of heuristics per tool. Grayed Pruning factors mark experiments that did not identify any relevant instructions.

Which tools and heuristics perform best depends on the use case. For `bzip2`, BinDiff gives the best pruning (see ⓐ), but only with the expansion heuristic. TurboDiff prunes less, but contrary to BinDiff, TurboDiff identifies all relevant instructions directly (see ⓑ). When only considering direct identification of relevant instructions, TurboDiff in general gives a higher recall than BinDiff (compare, e.g., ⓒ to ⓓ). For `png_beta`, BinDiff scores best (see ⓕ). BinDiff also works best for `soplex`, albeit with different heuristics (see ⓖ).

For `png_debian`, BinDiff's abstractions cannot identify any relevant instruction. TurboDiff identifies the changed immediate operands (see ⓒ and ⓔ), but only when using the appropriate heuristics. When those heuristics are applied for any other patch, they do not at all achieve the highest pruning.

For the minimal patch of `png_debian`, an attacker is better off with the binary code patch itself, which encodes precisely the semantic mutations to four immediate operands. The patch thus points the attackers directly to nothing more than the relevant instructions. This "shortcut" is only available,

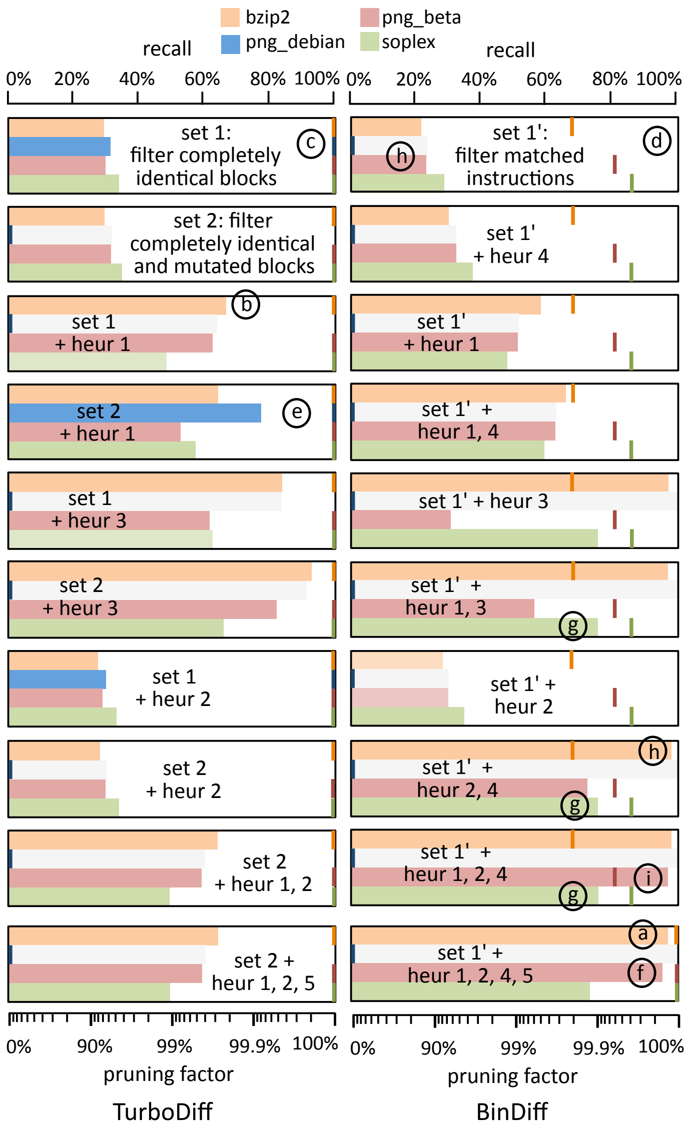


Fig. 4: Pruning factors (bars) and recalls (lines) obtained on undiversified binaries

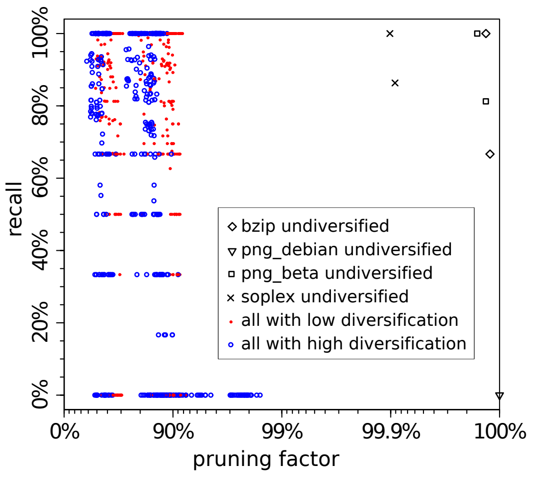
however, because all syntactic changes in this use case are semantic changes. When such a minimal security fix involving only changed constants is combined with other (non-related) fixes as in `png_debian`, the patch includes many more syntactic mutations, which prevents it from being used as a shortcut.

Considering only the combinations of tools and heuristics with recalls over 60%, the highest pruning factors obtained with BinDiff are 99.988% (ⓘ), 99.986% (ⓙ) and 99.909% (ⓔ). As the fractions of irrelevant instructions in those cases are 99.997%, 99.986%, and 99.923% resp., BinDiff proves to be able to prune more than 99.98% of all irrelevant instructions for those three use cases.

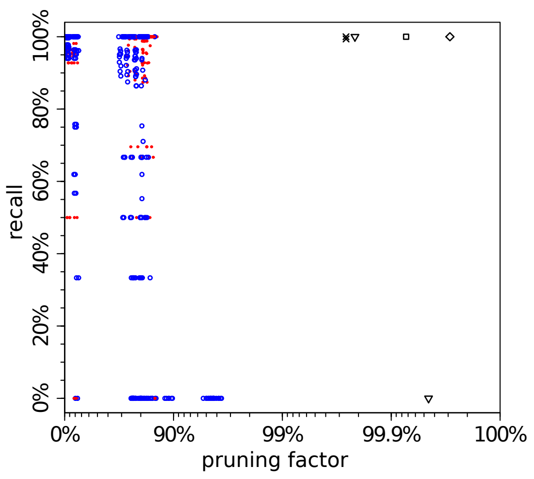
This demonstrates, for the first time, that for some types of patches and undiversified code, diffing tools and heuristics are indeed highly valuable cracker tools. For other types of patches, however, they are much less effective. Moreover, as a cracker does typically not know beforehand which types of patches have been applied, he will be hindered by not being able to fine-tune his heuristics.

## Diversification

To study the impact of diversification, we used the diversifier Proteus [1] that comes with the free and open Diablo link-time rewriting framework (http://diablo.elis.ugent.be). This tool supports a number



(a) results with BinDiff



(b) results with TurboDiff

Fig. 5: Comparison of results on diversified and undiversified binaries

of standard code generation, optimization and obfuscation techniques, but rather than optimizing a performance or software protection objective, the diversifier applies the transformations in a stochastic manner using a pseudo-random number generator (PRNG). Different versions of a binary can be generated simply by feeding the PRNG with different seeds. To trade-off the level of diversification with the overhead introduced by this stochastic application of transformations, the user can select the probabilities with which transformations are applied. When generating two program versions with all transformation probabilities set to 0.5, Proteus generates the most diverse binaries.

Some important aspects of the diversification are that the mapping between program fragments in two diversified binaries is an n-to-m mapping, that the shapes of CFGs become heavily mutated, that direct control flow transfers are hidden behind indirect ones, and that the code layout of a program cannot be relied upon to reconstruct functions. These properties directly target the matching heuristics in the diffing tools that rely on function CFG similarity and on the rather common assumption of one-to-one mappings.

It is important to note that the diversifying transformations we used are very similar to standard code transformations in compilers. For example, the applied transformations are already available or can easily be implemented in LLVM (`http://llvm.org`, a compiler used and trusted in industry. Diversification is hence not fundamentally less reliable than the use of a compiler in the first place.

To study the trade-off between security-wise effectiveness and performance-wise efficiency, we present results for binaries diversified at two levels of diversification. Experiments with other settings confirm the trends presented here. For the four use cases and two diversification levels, we generated 10 pairs of an unpatched and a patched binary, using 80 different PRNG seeds. We then applied 40 combinations of tools and heuristics on the $8 \times 10$ pairs of patched and unpatched program versions, for a total of 3200 diffing attempts.

In Figure 5, the results of the 1600 attempts with BinDiff and TurboDiff are plotted next to the Pareto-optimal results from Figure 4. With the right heuristics, which differ from patch to patch, the tools can still retrieve all relevant instructions from the diversified binaries. However, they can only do so at pruning factors of around 90%. The amount of code still needing manual analysis by the cracker has thus grown 100-fold. It can also be observed that higher levels of diversification do not offer a lot of additional protection.

Table 1 shows that our diversification comes with considerable overhead. While the binaries only grow with 20% with low diversity, the patches that would be distributed typically become between 1 and 2 orders of magnitude larger. For the small `png_debian`, the patch becomes even 1690 times bigger. Furthermore, our stochastic diversification results in considerable slowdowns, ranging from 20% to 39% for low diversification. For higher diversification, the overhead becomes even higher.

## Resilience

Our experimental results hint that when applied to diversified software, diffing tools become less effective. Attackers can try to work around this problem, however, by not diffing the original and patched binaries, but by diffing normalized versions instead, in which complex code constructs and variations are replaced by simpler, standardized versions. For example, after collecting traces of a program, tools like Ariadne (`http://ariadne.group-ib.ru/`) can replace obfuscated, indirect control flow by unobfuscated, direct counterparts. This will help IDA Pro in determining the functions and minimize the number of syntactic differences. Diablo itself has already been used to perform trace-based deobfuscation [10] and tools like DynInst (`http://www.dyninst.org/`) can collect traces even in the presence of advanced anti-debugging techniques. So nothing seems to stop attackers from undoing most if not all of the diversification before diffing the programs.

They do face a couple of major hurdles, however. First, when diversification would be implemented in a modular compiler like LLVM, it can build on the randomized enabling and disabling of all available code transformations. As many transformations depend on each other (sometimes involving phase-ordering issues) and on the availability of high-level semantic information (such as type information) which is not available in binary code, fully deoptimizing a binary program to normalize it is impossible, as is fully optimizing it. So it is doubtful that complete or even close to complete normalization is possible.

Secondly, the normalization can be expected to be sensitive to the precise form of diversification used. For example, an attacker in search might try to remove opaque predicates from both program versions by eliminating conditional branches that are only executed in one direction on his set of inputs. But since he does not yet have an exploit input to triggers new input-validity checks, the corresponding branches will also be eliminated. By parameterizing the diversification, the software vendor might be able to force the attacker to retune his tools and heuristics for every released patch or to make them interactive. In both cases, this would increase the attacker's effort and shorten his window of opportunity.

## Conclusions

Our experiments have shown that software diversification can significantly reduce the effectiveness of diffing tools taken for granted by patch-based exploit developers. Diversification thus offers the potential to shorten their window of opportunity and potential gains.

Our form of stochastic diversification comes with high overhead, however, so for now we consider diversification an effective, but not necessarily efficient protection against patch-based attacks. Whether or not it is worth the overhead will depend on the criticality of a patch. When a white-hat security researcher contacts a developer about an exploitable zero-day vulnerability, it can be critical to fix the vulnerability without exposing it publicly by means of an all too obvious patch. When some bug has been known publicly for a long time but no exploit has ever been constructed, there will be less need to protect a patch.

In the continuing software protection arms race, its is the defender's job to keep up with the attackers. Whenever one of them progresses, an evaluation like in this paper will have to be repeated to reevaluate the position of both players. We therefore believe that the methodology presented here will prove useful far beyond our evaluation of today's tools. In the near future, we plan to work on the trade-off between overhead and protection by using a more controlled, auto-tuning approach driven by feedback from the diffing scripts, as well as on the incorporation of normalization tools on the attacker's side.

## References

[1] Anckaert, B. *Diversity for Software Protection.* PhD thesis, Ghent University, 2008.

[2] Barthen. [WoW] [3.0.9] Symbolic info. Forumpost at `http://www.mmowned.com/forums/world-of-warcraft/bots-programs/memory-editing/219320-wow-3-0-9-symbolic-info.html`, 2009.

[3] Brumley, D., Poosankam, P., Song, D., and Zheng, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symposium on Security and Privacy* (2008).

[4] Collberg, C., and Nagra, J. *Surreptitious Software: Obfuscation, Watermarking, and Tamper-proofing for Software Protection.* Addison-Wesley Professional, 2009.

[5] Core Security Technologies. Windows SMTP service DNS query id vulnerabilities. CoreLabs Security Advisory, 2010.

[6] Eagle, C. *The IDA Pro Book*, 2 ed. No Starch Press, 2011.

[7] Economou, N. Microsoft Virtual PC: The hyper-hole-visor bug & MS10-048: Win32k window creation vulnerability (CVE-2010-1897), 2010.

[8] Harris, S., Harper, A., Eagle, C., and Ness, J. *Gray hat hacking: the ethical hacker's handbook.* McGraw-Hill, 2008.

[9] Lee, B., and Yeong Jang, Y. J. Exploit shop website.

[10] Madou, M., Anckaert, B., De Sutter, B., and De Bosschere, K. Hybrid static-dynamic attacks against software protection mechanisms. In *DRM '05: Proceedings of the 5th ACM workshop on Digital rights management* (2005), pp. 75–82.

[11] Manning, C. D., and Schütze, H. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[12] Moore, H. Exploiting IIS via HTMLEncode (MS08-006). Blogpost, 2008.

[13] Oh, J. Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. In *BlackHat USA* (2009).

[14] Protas, A., and Manzuik, S. Skeletons in Microsoft's closet - silently fixed vulnerabilities. BlackHat Europe, 2006.

[15] Sotirov, A. Reverse engineering Microsoft binaries. CanSecWest, 2006.

**Bart Coppens** is a PhD student in the Computer Systems Lab at Ghent University, Belgium. His research focuses on software protection, in particular on software diversity and side-channels.

**Bjorn De Sutter** is a professor in the Computer Systems Lab at Ghent University. His research focuses on compiler techniques for diverse applications, including software protection.

**Koen De Bosschere** is a professor in the Computer Systems Lab at Ghent University and the coordinator of the European FP7 network-of-excellence HiPEAC.