Feedback-Driven Binary Code Diversification

Bart Coppens, Computer Systems Lab, Ghent University
Bjorn De Sutter, Computer Systems Lab, Ghent University
Jonas Maebe, Computer Systems Lab, Ghent University

As described in many blog posts and in the scientific literature, exploits for software vulnerabilities are often engineered on the basis of patches. For example, "Microsoft Patch Tuesday" is often followed by "Exploit Wednesday" during which yet unpatched systems become vulnerable to patch-based exploits. Part of the patch engineering includes the identification of the vulnerable binary code by means of reverse-engineering tools and diffing add-ons. In this paper, we present a feedback-driven compiler tool flow that iteratively transforms code until diffing tools become ineffective enough to close the "Exploit Wednesday" window of opportunity. We demonstrate the tool's effectiveness on a set of real-world patches and against the latest version of BinDiff.

Categories and Subject Descriptors: D.2.0 [**Software Engineering**]: General—*Protection mechanisms*; D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Optimization*

General Terms: Experimentation, Measurement, Security

Additional Key Words and Phrases: compiler transformations, software diversity, binary diffing, program matching, patches

1. INTRODUCTION

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 vulnerabilities being patched, and some patched vulnerabilities might not be mentioned at all [Core Security Technologies 2010].

So when crackers get their hands on the binary patches, they start inspecting them in preparation of *Exploit Wednesday*, the cracker's window of opportunity to target unsuspicious users that did not immediately apply the update. In some cases, crackers can also target users that did apply it immediately, but that were left vulnerable because a fix was not complete [Economou 2010]. With the help of so-called diffing tools like Darungrim and BinDiff, the crackers set up collusion attacks in which they compare the binary code before and after the patch to identify the fixed code fragments and the applied fixes, to determine the vulnerabilities closed by those fixes, and ultimately to devise actual exploits for those vulnerabilities.

Similar collusion attacks can be used to identify the code that implements important new functionality in major 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 crackers can identify the corresponding parts in consecutive program versions, they can more easily reuse their existing reverse-engineering knowledge. This knowledge can consist of informal insights, but also of

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

more formal information. For example, 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 [Barthen 2009]. And in yet another related attack scenario, collusion attacks try to extract information such as cryptographic keys from differences between program versions [Boneh and Shaw 1998; Ergun et al. 1999].

Brumley et al [Brumley et al. 2008] demonstrated that sometimes exploits can be devised without any human analysis or understanding of the patched code. All their attack needs is an accurate identification of the modified instructions. On that basis, they apply constraint solving techniques on the program inputs to generate attacks fully automatically.

All published attacks that we are aware of similarly build on the assumption that expert crackers assisted by diffing tools can easily identify the relatively few relevant differences between unpatched and patched binaries. Many authors simply do not even consider it worthwhile to discuss how they identify the patched code fragments. Others discuss it very briefly. For example, Protas and Manzuik briefly describe their use of BinDiff to analyze undiversified Microsoft patches for Windows [Protas and Manzuik 2006]. Brumley et al mention their use of the e-Eye Binary Diffing Suite (EBDS) to analyze the syntactic difference between the two binaries [Brumley et al. 2008]. Oh observes that the engineering of most security exploits starts with the manual or automatic analyses of the differences created by security patches, and briefly describes how EBDS and Darungrim can be used to analyze those differences [Oh 2009]. Many other security researchers, hackers, and hobbyists tell a similar story, implicitly assuming that spotting the relevant differences is trivial with the existing tools [Loveless 2006; Varghese 2008; Moore 2008; Sotirov 2006; Lee and Jang 2012; Walia 2011; Frijters 2010; Slawlerguy 2008; Johnson 2011; Harris et al. 2008].

In our previous work [Coppens et al. 2012], we evaluated the effectiveness of four automated diffing tools to reduce the manual code investigation effort required from attackers. We concluded that the tools in combination with scriptable heuristics effectively prune over 99.9% from the attacker's search space, thus limiting the required manual investigation effort to less than 0.1% of the code. We also demonstrated that software diversification, with which we mean the transformation of code fragments that do not strictly need to be changed to implement a patch, can reduce the effectiveness of the diffing tools with a factor 100. The attacker then has to manually search about 10% of the code for changes, which significantly reduces his window of opportunity.

The factor 100 reduction in pruning effectiveness we obtained in our previous work, however, came at a high price of up to 39% execution slowdown. The main culprit of this unacceptable overhead was the uncontrolled manner (based only on a pseudo-random number generator and on profile information) in which the compiler transformations were applied to generate the necessary diversity.

In this paper, we present a novel, iterative, feedback-driven compiler diversification approach. Initially, the diversifier only applies diversifying transformations with near zero overhead to the patched program in order to thwart the diffing tools. We then run those diffing tools on the generated binary to diff it against the original (unpatched) binary, and feed the result back to the diversifier for the next iteration. In that iteration, the diversifier applies more diversifying transformations, but those are only applied to the fragments that were still matched correctly by the diffing tool in the previous iteration. In the following iterations, the diversifier will then gradually start to apply more transformations that potentially introduce more overhead, but those transformations are only applied to code fragments for which the lower-overhead transformations proved to be insufficiently effective. *With this approach, we are able to make the diffing tools almost completely useless, without introducing significant performance overhead.* We demonstrate this on several real-world applications and real-world security patches.

The remainder of this paper is structured as follows. Section 2 discusses in more detail how diffing tools aid attackers to exploit vulnerabilities on the basis of released binary code

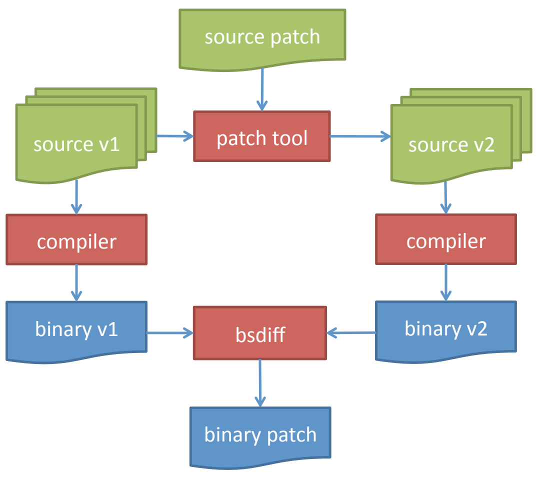


Fig. 1. Standard development of successive software versions.

patches. Section 3 present our new feedback-driven diversification approach to thwart those diffing tools. This approach is evaluated on real-life case studies in Section 4. Section 5 discusses the related work, after which the pros and contras of the approach are discussed in Section 6. Finally, conclusions are drawn in Section 7.

2. DIFFING TOOLS FOR PATCH-BASED ATTACKS AND DIVERSIFICATION

In this section, we first discuss the relation between source code patches and binary code patches and the information they contain. We then discuss how diffing tools help attackers in bridging the gap between the two, after which we focus on BinDiff ([zynamics 2012; Flake 2004; Dullien and Rolles 2005]), the particular tool we used to evaluate our approach.

2.1. Source vs. Binary Patches

To understand why and how attackers use diffing tools to engineer attacks on vulnerabilities fixed in released software updates, it is important to understand the relation between the released updates and the vulnerabilities. This relation is depicted in Figure 1. At the top of the figure, a source code patch fixes vulnerabilities in some program version 1. This patch can be applied to the original source code with, e.g., the open-source `patch` command-line tool. Besides fixes to security vulnerabilities, the patch might also include other changes, such as optimizations and extended functionality. For an Exploit Wednesday attacker, this patch would be extremely useful, as it almost directly leads him the vulnerable code fragments.

Attackers often don't get to see the source code patch, however. Instead they only get the binary of the original software v1, along with the binary patch that enables them to update their v1 into v2. As shown in the lower half of Figure 1, this binary patch is generated by the developer from the compiled binaries v1 and v2, for example by means of the open-source `bsdiff` [Percival 2003] command-line tool.

For several reasons, this binary patch is not nearly as useful to attackers as the source code patch would have been:

(1) The patch identifies code changes on binary code, which is a much lower abstraction level than source code. After having identified the locations in the binary code that were patched, the attacker will still have to reverse-engineer the code and the code changes.
(2) Because of two reasons, small source code patches typically result in significantly larger binary code patches:

(a) In source code, most references to code or data are symbolic, e.g., by means of procedure names or variable names. Those do not change globally because of a local source code patch. In binary executables, however, most references are encoded by means of absolute and relative addresses. As soon as code or data is inserted or removed by a software patch, it typically results in many small changes to hard-coded constants spread throughout the binary code. Typically, these changes occur in immediate operands and literal data pools.

(b) Global compiler optimizations can cause local patches to have much wider effects on the binary code. For example, a source code patch that corresponds to a local modification in the data dependence graph (DDG) of a procedure might result in very different register allocation and code scheduling of otherwise unrelated data flow. It might also result in loop unrolling being disabled or enabled. Similarly, small changes to the control flow graph (CFG) of a procedure can result in very different code layout. And small changes to a procedure's size can make the compiler switch between not inlining, full inlining, partial inlining, or cloning the procedure at some of its callers. The code in figures 7(c) and 8(c) illustrates this.

In general, when the semantics of a program are changed by patches at specific source code locations, the compiled binary will incur semantic code mutations at corresponding locations, but in addition, many smaller or bigger syntactic mutations will typically be spread throughout the binary.

So in summary, before developing an exploit, an attacker will have to reverse-engineer the vulnerable code fragments and the patches on those fragments. To minimize that effort, he will try to identify the most promising fragments by pruning the purely syntactic mutations from the binary patch. This is where diffing tools such as BinDiff[1], BinaryDiffer[2] (a variant of Darungrim[3], which in turn is part of EBDS[4]), PatchDiff2[5], and TurboDiff[6] come into play.

2.2. Diffing Tools

Diffing tools try to find matching and differing code fragments in two program versions at a higher abstraction level than the byte level at which tools like `bsdiff` operate. Figure 2 depicts their use by attackers.

From the original binary software and the patch, the attacker generates the patched binary version in the same way any ordinary user would do, for example by means of the `bspatch` counterpart of `bsdiff`. Then the attacker feeds the two binary versions to a diffing tool. These tools are typically plug-ins for the IDA Pro[7] disassembler. IDA Pro is an interactive disassembler that offers users a graphical interface to study binary code. It not only gives users an editable disassembly of binaries, but it can also build and show their CFGs and call graphs (CGs). Furthermore, it is fully extensible through plug-ins such as the diffing tools. These plug-ins can then use the disassembly and analyses done by IDA Pro in their matching techniques.

The results are presented to the attacker in different ways in the different tools. They all have in common that the attacker can easily browse the graphs of completely or partially matching procedures side by side in a GUI as shown in Figure 2. Through colors that indicate the level of matching/difference, the attacker can quickly identify the most promising code

[1]`http://www.zynamics.com/bindiff.html`
[2]`https://code.google.com/p/binarydiffer/`
[3]`http://www.darungrim.org/`
[4]`http://www.eeye.com/resources/security-center/research/tools/eeye-binary-diffing-suite-ebds`
[5]`https://code.google.com/p/patchdiff2/`
[6]`http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=turbodiff`
[7]`http://www.hex-rays.com/products/ida/`

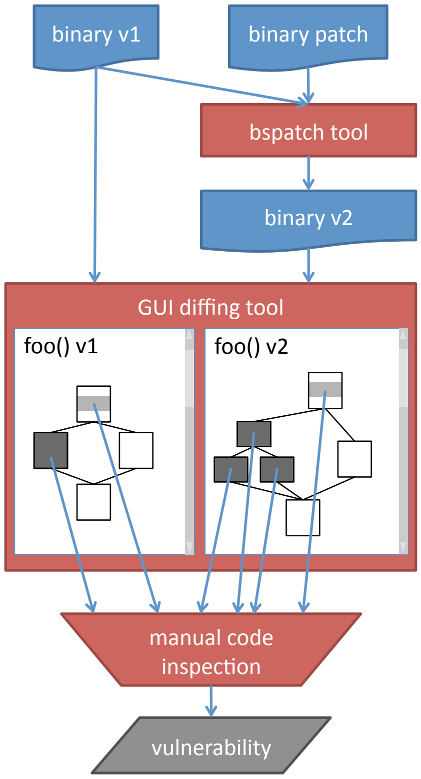


Fig. 2. Tool flow for patch-based attacks.

fragments. Because the matching techniques are applied at the CFG and CG level, they make abstraction of the first cause of syntactic differences discussed above. As such, they provide excellent search space pruning for the attackers.

Besides in the code matched but not fully identical fragments marked in light and dark gray in Figure 2, relevant semantic changes might also occur in procedures for which BinDiff did not find a matching counterpart at all, or even in code that was not recognized as code by IDA Pro, and that hence did not get disassembled and included in the CFGs. Obviously an attacker will also have to consider those fragments. So an attacker will have to investigate all code in unmatched procedures, as well as all differing code in (partially) matched procedures.

Our previous work evaluated the effectiveness of BinDiff, BinaryDiffer, TurboDiff and PatchDiff2 as tools for patch attackers [Coppens et al. 2012]. The main observations and general conclusions were the following:

— All tools first try to match corresponding procedures in both programs, after which TurboDiff, BinDiff and BinaryDiffer also try to match basic blocks within matched procedures. Only BinDiff then tries to match instructions within matched basic blocks.
— BinDiff is overall more efficient than the other tools in matching procedures, basic blocks, and instructions, as well as in pruning non-interesting code.
— However, BinDiff sometimes prunes too much code. When a patch involves the simple replacement of one constant value by another, BinDiff's heuristics (at their specific abstraction level) sometimes overlook this difference. One such case occurs with the `png_debian`

off-by-one patch presented in Section 4.1 and visualized in figures 7(b) and 8(b). PatchDiff2 and BinaryDiffer suffered from this problem as well.

— All diffing plug-ins suffer from IDA Pro's problem to handle unconventional code. This is particularly the case for procedures of which the code is not stored contiguously and for obfuscated code. Obviously, when IDA Pro cannot partition the code into procedures correctly, its plug-ins face difficulties matching procedures.

In Section 3, we will present an approach that aims for making the matching heuristics of diffing tools ineffective. Given the above observations, BinDiff is the best tool to evaluate the effectiveness of this approach. In the remainder of this paper, we therefore focus on BinDiff.

2.3. BinDiff

BinDiff is a commercial, closed-source tool from Zynamics, now Google. Its high-level internal operation is described in its manual [Zynamics 2012] and in a couple of publications [Dullien and Rolles 2005; Flake 2004]. Most important in the context of this paper is the procedure matching strategy in BinDiff.

For all procedures in both programs to be matched, numerous signatures are computed based on their names (if not stripped from the binaries), on the binary code in the procedure bodies, on properties of their CFGs (such as number of loops in them, number and topology of the edges, number of basic blocks, etc), on their (nested) callers and callees, and on the strings they reference. In both programs, the sets of procedures are first partitioned on the basis of the most discriminative, stronger signature(s). When a singleton partition (i.e., one procedure) with the same signature(s) occurs in both versions, those two procedures are considered a match. When partitions in both programs with matching signatures consist of more than one procedure, BinDiff will iteratively partition those partitions into smaller subpartitions on the basis of more (ever weaker) signatures until singleton subpartitions occur. At any iteration, all procedures without matching signatures in the other program version are grouped into the "unmatched" partition. This partition is repartitioned iteratively in the same way as non-singleton partitions with matching signatures.

The main result of BinDiff then is a list of matched functions in order of decreasing matching quality. For each pair of matching procedures, BinDiff computes all possible signatures. The matching quality, which is essentially a confidence metric, is then computed on the basis of the number of signatures that are identical for both versions and on the strengths of those identical signatures. Furthermore, BinDiff reports the signature on the basis of which procedures were matched, i.e., the signature on the basis of which singleton partitions were obtained during the iterative subpartitioning. It is this feedback that will drive the tool we present in the Section 3, as shown in Figure 3.

Besides the above outputs, BinDiff also reports the similarity between two matched functions. An attacker will use all this information to select matched procedures for manual inspection: procedures that BinDiff believes to correspond with high quality but that are not completely similar likely feature promising code fragments. The attacker can simply click through to them and observe the similarities and the differences as shown in Figure 2.

3. FEEDBACK-GUIDED DIVERSIFICATION

In this section, we first present an overview of our approach. This is followed by a description of the code transformations that are applied to generate diversification and a description of the decision logic of their application.

3.1. Overview

Figure 3 shows the tool flow of our approach. Compared to the tool flow of Figure 1, we have replaced the original compiler toolbox with one that can apply diversification. This

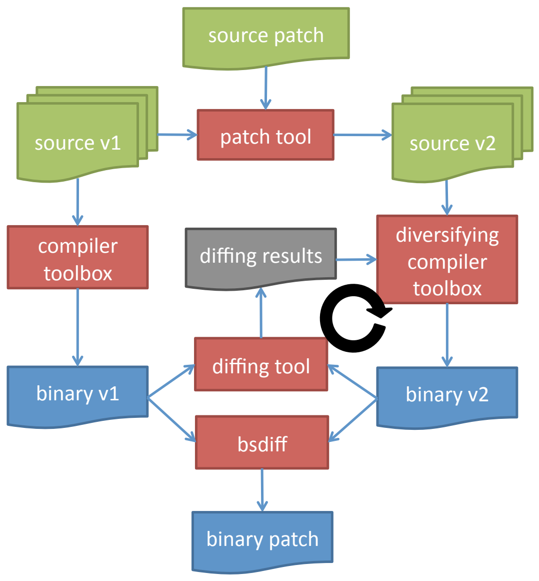


Fig. 3. Iterative tool flow for generating protected patches..

toolbox is applied iteratively on the basis of feedback obtained from one or more diffing tools that model the attack tools against which one tries to defend.

In our proof-of-concept implementation, our diversifying compiler toolbox consists of the standard GCC 4.6 compiler plus an evolution of the binary code diversification tool Proteus [Anckaert 2008] that comes with the free and open Diablo link-time rewriting framework (http://diablo.elis.ugent.be). Proteus supports a number 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.

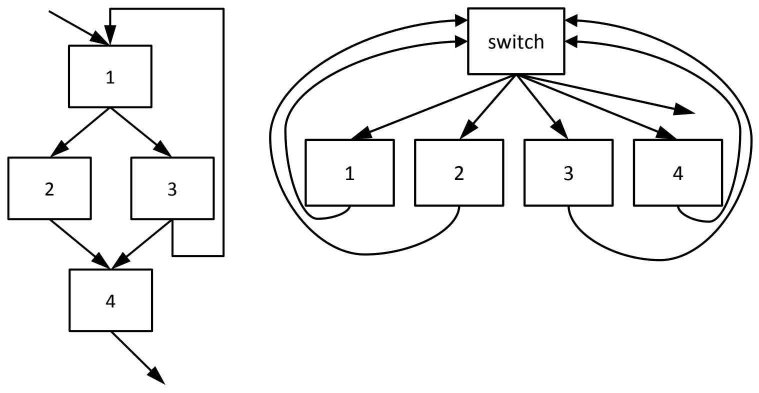
In our evolution of Proteus, the application of the transformations is not a pure stochastic process. Instead its decision logic is based on the PRNG, on profile information obtained on the original binary, on the feedback obtained from the diffing tools, and on the number of iterations already performed.

3.2. Diversifying Transformations

Our diversifier can apply combinations of five transformations.

Code Layout Randomization. The code diversifier randomizes the order in which basic blocks chains (i.e., sequences of basic blocks chained together in fall-through paths) are placed in the executable's code section. As a result, most procedure bodies are not stored contiguously. This complicates IDA Pro's partitioning of the disassembled code in procedures and its construction of the CG and the CFGs.

The performance impact of chosing a different order of the basic block chains is minimal. Because chains of frequently executed basic blocks are not split by introducing additional unconditional branch instructions, only the instruction cache behavior can deteriorate.



(a) Original CFG (b) Flattened CFG

Fig. 4. Control flow flattening

Partial Control Flow Flattening. Control flow flattening replaces an original CFG by one where all original basic blocks have the same predecessor and successor [Wang et al. 2001]. The transformation is illustrated in Figure 4. Semantic equivalence is guaranteed by inserting a redirection variable guiding the execution through the switch statement.

In the original application context of flattening, being control flow obfuscation, as much as possible control needs to be flattened, in particular of the most interesting code, i.e., the code that is executed (frequently). Hence outside of this work, flattening is typically applied on whole procedures at a time.

Our diversification tool can flatten parts of procedures, hence the name partial flattening. The reason is that flattening in our context aims not for obfuscation, but for thwarting diffing tool heuristics based on CFG topology properties. Changing parts of the CFGs, such as the coldest parts, suffices for that purpose.

By applying flattening to cold code only, its impact on performance can be minimized.

Conditional Branch Flipping. To thwart some very simple matching heuristics, much simpler CFG transformations suffice. Conditional branch flipping flips the branch-taken and fall-through edge following a conditional branch instruction in a CFG, while at the same time inverting the branch condition, e.g., from branch-if-greater to branch-if-less-or-equal.

This transformation has almost no influence on performance, in particular when limited to cold code only. The influence on code size is minimal as well.

Two-way Opaque Predicate Insertion. As illustrated in Figure 5, two-way opaque predicate insertion involves the duplication of (part of) a basic block, and insertion of a random branch condition [Collberg et al. 1998]. The duplicate blocks can be transformed independently by later transformations. This transformation targets very simple matching heuristics such as instruction counts. Its impact on performance can be limited by applying the transformation to cold code only.

Branch Function Insertion and Call Function Insertion. For each direct control transfer, be it a jump or a fall-through path, we can redirect it through a branch function [Linn and Debray 2003]. Branch functions are functions that do not return to their caller; instead they transfer control to a different address computed from the return address and an offset passed to the branch function as a parameter. So a direct jump or fall-through can be replaced by a call and an indirect jump based on the call's arguments. Figure 6(a) shows an unconditional jump which is transformed into a call to the branch function in Figure 6(b).

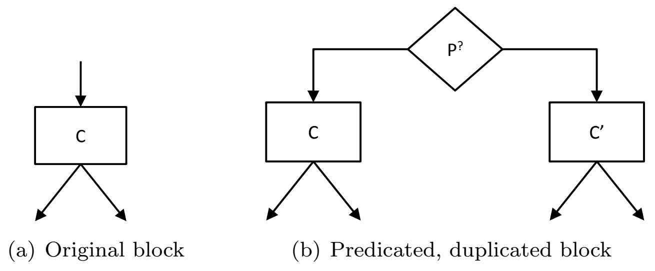


Fig. 5. Predicating a basic block by a two-way opaque predicate

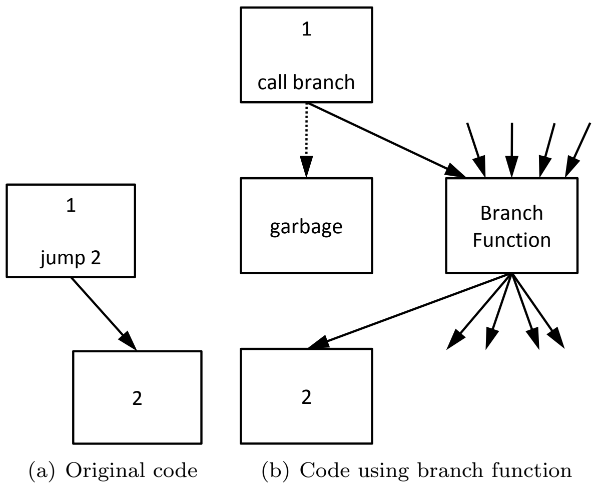


Fig. 6. Branch function insertion

The main reason for inserting branch functions is to split procedure bodies in parts that are no longer connected through direct control flow transfers. This thwarts the CFG construction in IDA Pro, and hence the matching in the diffing plug-ins. Branch functions can also be used to replace direct procedure calls by indirect control flow. It suffices to replace the call by a push of the return address on the stack and a call to the branch function to transfer control to the callee. Branch functions can therefore thwart both CFG-based and CG-based matching heuristics.

In the remainder of this paper, we use the term *Branch Function Insertion* to refer to the replacement of direct jumps or fall-through paths, and the term *Call Function Insertion* to refer to the replacement of direct calls. On the case of Branch Function Insertion, we sometimes specify where it needs to be applied to achieve different goals. For example, by inserting a branch function in the entry block of a function, most of its body becomes disconnected from that entry point. By inserting a branch function before a call, that call is not removed, but it becomes disconnected of the preceding code.

Branch functions are clearly expensive in terms of performance overhead when applied to frequently executed code. Besides the overhead of additional instructions, the indirect jump out of the branch function is typically not predicted well by the processor's branch predictor.

3.3. Transformation Selection

The selection of diversifying transformations to be applied is based on a set of rules encoded in a rule set table. The table used for the experimental evaluation in Section 4 is depicted in Table I.

Each row in the table specifies a *necessary condition* to apply a transformation. A row with signature S (we refer to the BinDiff manual [Zynamics 2012] for a detailed description of the exact signatures) , relative weight[8] W, iteration I and transformation T specifies that in any iteration $i \geq I$, transformation T can be applied to basic blocks with a relative weight $w \leq W$ in procedures reported by the diffing tool to have been matched on the basis of signature S in iteration $i-1$.

For example, the first row in Table I indicates that conditional branches may be flipped in any iteration, in any procedure matched by means of the "Hash Matching" signature, independent of the weight of the block of the branches. The third row indicates that from the first iteration on, two-way predicates can be inserted in or before basic blocks that have weight zero (i.e., that are not executed) in procedures that were matched on the basis of the "Edges Flowgraph" signature. It is clear that gradually transformations with higher overhead will be considered, and that their application onto ever hotter blocks is considered.

During the successive iterations, a diversification strategy is built for each procedure. This strategy is another table that stores each applied transformation, the program point where it was applied, the iteration in which it was applied, the reason why it was applied (i.e., the BinDiff signature targeted with it, which comes from the rules table), and the PRNG seed that was used for selecting that transformation. An example table for a function `f()` is shown in Table II.

Initially, i.e., after iteration zero in which only the code layout is randomized, each procedures strategy is empty. Then during each iteration, three actions can be performed on the strategy: it can remain identical, it can be extended, or it can be adapted. Assume we are in iteration $i \geq 1$.

(1) When a procedure is not matched by BinDiff according to the feedback of iteration $i-1$, the strategy remains untouched. This happens when the strategy succeeded in thwarting BinDiff completely for this procedure. It implies that we will apply exactly the same transformations in iteration i.
(2) When a procedure is matched by BinDiff with some signature s according to the feedback of iteration $i-1$, and that s is not the same as the last signature s' in the strategy, this implies that the strategy was successful in thwarting matching based on s', but not in thwarting matching based on s. In this case, the strategy is extended: From all transformations that meet the necessary conditions as specified by the rule table, one is selected randomly and appended to the strategy. By construction, this will be a transformation that targets signature s.
(3) When a procedure is matched by BinDiff with the same signature s as already occurred in the last rows of the existing strategy, this implies that the strategy was not successful for this procedure. We then remove the transformations from the last iteration from the strategy, and replace them with a new selection of transformations. This selection happens on the basis of another random seed. Furthermore, we now select one more transformation than we selected in the previous iteration. Moreover, the set of applicable transformations can have become bigger because new rules become applicable in later iterations.

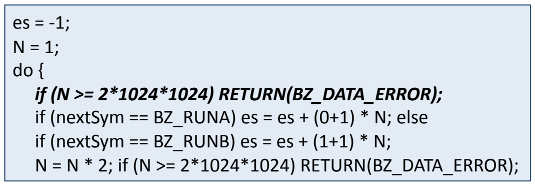
[8]The weight of a block is its execution count given a training profile, multiplied by the number of instructions in the block. The total weight of the program equals the summed weights of all blocks. The relative weight of a block is its weight divided by the total program weight.

Table I. Diversification rule set

Signature	Weight	Iteration	Transformation
Hash Matching	1.00000	1	Conditional Branch Flipping
Edges Flowgraph	1.00000	1	Conditional Branch Flipping
Edges Flowgraph	0.00000	1	Two-way Opaque Predicate Insertion
Edges Flowgraph	0.00005	5	Two-way Opaque Predicate Insertion
Edges Flowgraph	0.00005	6	Partial Control Flow Flattening
Edges Callgraph	0.00000	1	Branch Function Insertion (anywhere)
Edges Callgraph	0.00005	5	Branch Function Insertion (anywhere)
Edges Callgraph	0.00000	1	Two-way Opaque Predicate Insertion
Edges Callgraph	0.00005	5	Two-way Opaque Predicate Insertion
Instruction Signature	1.00000	1	Conditional Branch Flipping
Call Sequence	0.00000	5	Call Function Insertion
Call Sequence	0.00005	7	Call Function Insertion
Call Sequence	0.00015	9	Call Function Insertion
Call Sequence	0.00000	11	Branch Function Insertion (before calls)
Call Sequence	0.00005	13	Branch Function Insertion (before calls)
Call Reference	0.00000	5	Call Function Insertion
Call Reference	0.00005	7	Call Function Insertion
Call Reference	0.00015	9	Call Function Insertion
Call Reference	0.00000	11	Branch Function Insertion (before calls)
Call Reference	0.00005	13	Branch Function Insertion (before calls)
Call Sequence (Exact)	0.00000	7	Call Function Insertion
Call Sequence (Exact)	0.00005	9	Call Function Insertion
Call Sequence (Exact)	0.00015	11	Call Function Insertion
Call Sequence (Exact)	0.00000	13	Branch Function Insertion (before calls)
Call Sequence (Exact)	0.00005	15	Branch Function Insertion (before calls)
Call Sequence (Topology)	0.00000	7	Call Function Insertion
Call Sequence (Topology)	0.00005	9	Call Function Insertion
Call Sequence (Topology)	0.00015	11	Call Function Insertion
Call Sequence (Topology)	0.00000	13	Branch Function Insertion (before calls)
Call Sequence (Topology)	0.00005	15	Branch Function Insertion (before calls)
Hash Matching	0.00000	2	Branch Function Insertion (in entry blocks)
Edges Flowgraph	0.00000	2	Branch Function Insertion (in entry blocks)
Edges Callgraph	0.00000	2	Branch Function Insertion (in entry blocks)
Instruction Signature	0.00000	2	Branch Function Insertion (in entry blocks)
Call Sequence (Exact)	0.00000	2	Branch Function Insertion (in entry blocks)
Call Sequence (Topology)	0.00000	2	Branch Function Insertion (in entry blocks)
Call Sequence	0.00000	2	Branch Function Insertion (in entry blocks)
Call Reference	0.00000	2	Branch Function Insertion (in entry blocks)
String Reference	0.00000	2	Branch Function Insertion (in entry blocks)
Hash Matching	0.00005	3	Branch Function Insertion (in entry blocks)
Edges Flowgraph	0.00005	3	Branch Function Insertion (in entry blocks)
Edges Callgraph	0.00005	3	Branch Function Insertion (in entry blocks)
Instruction Signature	0.00005	3	Branch Function Insertion (in entry blocks)
Call Sequence (Exact)	0.00005	3	Branch Function Insertion (in entry blocks)
Call Sequence (Topology)	0.00005	3	Branch Function Insertion (in entry blocks)
Call Sequence	0.00005	3	Branch Function Insertion (in entry blocks)
Call Reference	0.00005	3	Branch Function Insertion (in entry blocks)
String Reference	0.00005	3	Branch Function Insertion (in entry blocks)

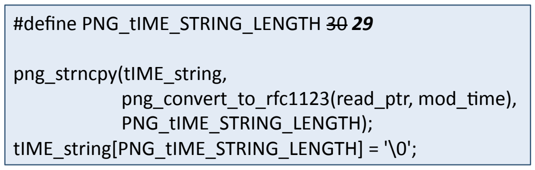
Table II. Diversification strategy for a procedure `f()`

Transformation	Program Point	Iteration	Signature	Random Seed
Conditional Branch Flipping	BBL 2356	1	Hash Matching	14562
Branch Function Insertion (anywhere)	BBL 2347	3	Edges Flowgraph	16382
Branch Function Insertion (anywhere)	BBL 2349	3	Edges Flowgraph	16382



```
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);
```

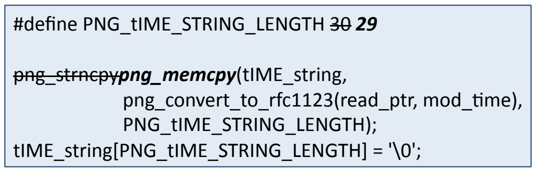
(a) bzip2 patch



```
#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. 7. Three of the four source code patches

Given these rules, the diversification strategy from Table II can be interpreted as follows: Assuming we are now after iteration 5, Conditional Branch Flipping succeeded in thwarting signature "Hash Matching" in iteration 1. From then on signature "Edges Flowgraph" became the target. Two transformations needed to be applied in thwart this signature, which was discovered in iteration 3. When all three transformations are applied in combination with code layout randomization, BinDiff is not able to match procedure `f()` anymore.

4. EXPERIMENTAL EVALUATION

In this section, we first discuss the case studies we evaluated our approach on. Next, we discuss some of the extensions to IDA Pro and BinDiff we implemented to overcome some of their limitations. We then analyze the effectiveness and the efficiency of our approach on the use cases.

4.1. Case Studies

The first patch on which we evaluated our approach, hereafter called `bzip2`, fixed vulnerability CVE2010-0405 in the program `bzip2` by inserting a validation check on an intermediate value as indicated in Figure 7(a). In the binary, this corresponds to a short instruction sequence being inserted as shown in Figure 8(a).

Our second patch is an off-by-one fix for vulnerability CVE-2008-3964 in the `pngtest` utility. The fix decrements a hard-coded value as shown in Figure 7(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 this patch resulted in four immediate instruction operands being replaced: in two similar fragments a constant operand 30 is replaced by 29 and the absolute address of `tIME_string[30]` is replaced by that of `tIME_string[29]`. One of those changed fragments is shown in Figure 8(b).

```
0x080538b4:   movl   $0xffffffff,0x60(%esp)
0x080538bc:   movl   $0x1,0x5c(%esp)
***0x080538c4:   cmpl   $0x1fffff,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. 8. Semantic changes in three of the four binary code patches

Table III. Case studies

Use case	Binary code size	Binary patch size
`bzip2`	407 kB	1.4kB
`png_debian`	514 kB	0.15kB
`png_beta`	538 kB	54.17kB
`soplex`	911 kB	14.26kB

In other distributions, this fix was part of a larger update from `libpng` 1.2.23-beta01 to beta02. In that larger update, which also contains a lot of patches not related to CVE-2008-3964, the code fragments using the changed constant were patched as shown in Figure 7(c). In addition to the changed constant, the call to `png_strncpy` is replaced by a call to `png_memcpy`. The compiler inlines that call and unrolls the loop responsible for copying the actual data in it, 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 8(c). The constant value 29 does therefore not occur in the patched binary anymore. We refer to this patch as `png_beta`.

Finally, we chose the SPEC benchmark program `soplex` as the target of a patch that replaces two (out of several more) calls to `quicksort` with calls to a newly added `mergesort`. This patch is called `soplex`.

We compiled and statically linked the original and patched source code on Linux with `gcc 4.6` at optimization level `-O3`. Table III shows the patched binary sizes as well as the sizes of the binary patches generated with the `bsdiff` tool that are typically distributed to the end-users. The three relatively large patch sizes indicate that those patches indeed involve many syntactic mutations as discussed in Section 2.1. The attacker's goal is therefore to weed those syntactic mutations out by means of BinDiff.

In the experiments we performed, we used the SPEC training inputs to collect profile information for `bzip2` and `soplex`. Whenever we report performance overhead, we used reference inputs.

4.2. Extensions to IDA Pro and BinDiff

IDA Pro and its plug-ins are interactive GUI tools, and that is how attackers normally use them. For example, when hackers load a binary into IDA Pro and have it disassembled, they will likely observe that not all code is disassembled by IDA Pro's standard recursive-descent disassembler [Eagle 2011]. They can then mark additional addresses in the code section of the binary and instruct IDA Pro to continue disassembling from those addresses.

With our approach, our main concern is not to thwart IDA Pro's disassembly process. Instead, we want to measure the effectiveness of IDA Pro and BinDiff in the hands of a real-world attacker that exploits IDA Pro's interactivity to circumvent disassembly issues.

To model such attackers, we engineered a script that instructs IDA Pro to continue disassembling code at additional addresses until the whole code section in the binary is disassembled. This script is invoked on each binary before it is fed to BinDiff. That way, all of our experiments model an expert attacker that is not fooled by static disassembly thwarting [Linn and Debray 2003].

4.3. Diffing Results

Using the rule set of Table I, we generated diversified binaries for our four use cases.

For `bzip2`, the chart in Figure 9 depicts the fraction of the code that BinDiff is able to match in different iterations. For the precise meaning of the different matching heuristics, we refer to the BinDiff manual [Zynamics 2012]. For each iteration, the left bar indicates the matches reported by BinDiff. Some of those matches, in particularly the ones based on low quality heuristics and signatures, are false positive matches, however. The right bars therefore show the fractions of instructions that were matched correctly. It is this fraction that is

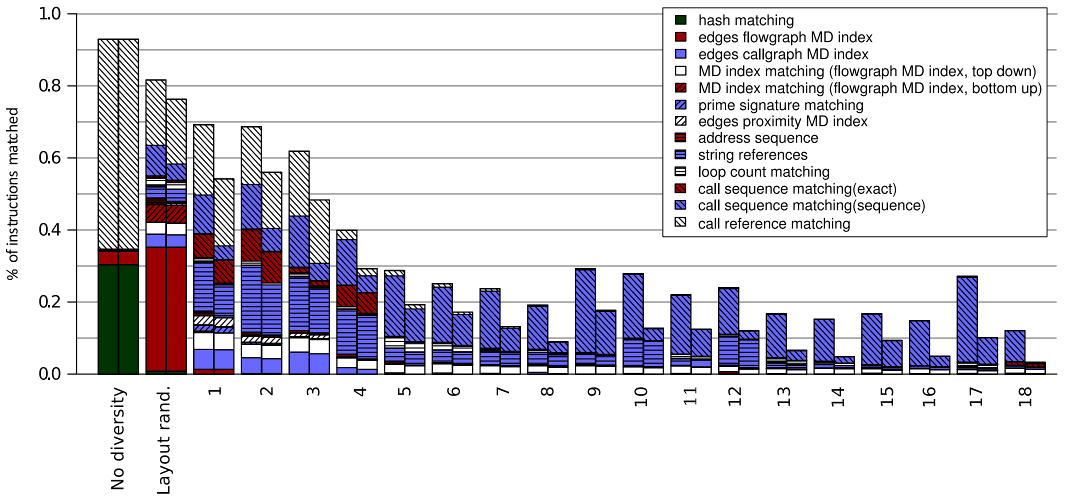


Fig. 9. Diffing results for the `bzip2` use case, indicating the decisive matching heuristics. For each point on the X-axis, the left bar shows the percentage of all instructions that Bindiff reports as having been matched. The right bar shows the percentage of instructions that it actually matched correctly.

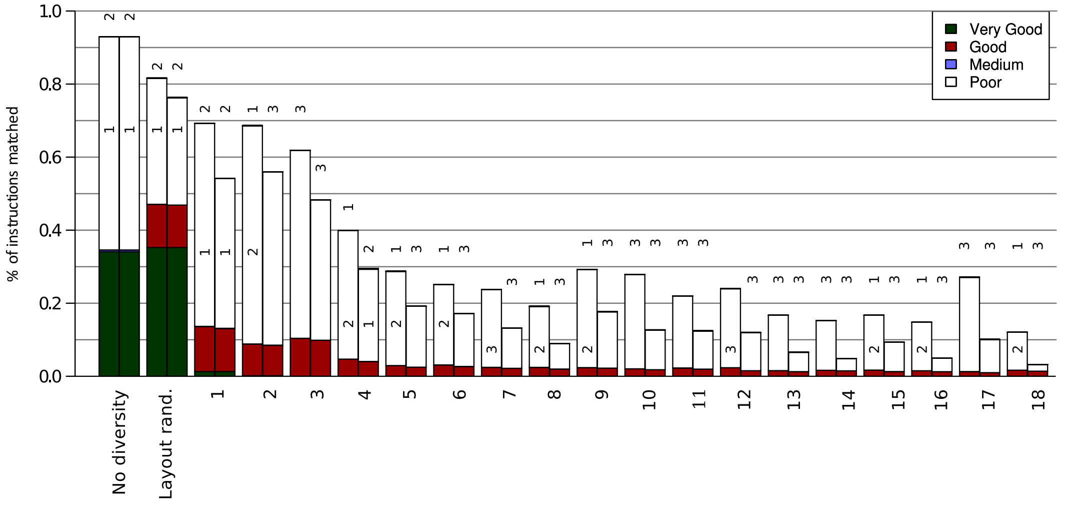


Fig. 10. Diffing results for the `bzip2`,indicating match quality of the decisive matching heuristics. The labels in the bars indicate the number of relevant instructions (i.e., that implement the actual security patch) that have been matched with the match quality indicated in the bar. Numbers outside the bar indicate the instructions of the patch that have not been matched.

most useful to an attacker. The chart in Figure 10 displays the same fraction, but this time with an indication of the match quality according to the BinDiff manual [Zynamics 2012]. The numbers above or in the different bars indicate how many of the relevant instructions (i.e., the bold italic instructions in Figure 8) are found in each category. Numbers above the bars indicate the number of those instructions that are in the unmatched part of the code, i.e., the part of the code within which BinDiff provides no help to the attacker at all[9].

[9]The total number of relevant instructions can vary from one iteration to the other when the relevant instruction sequences are diversified itself.

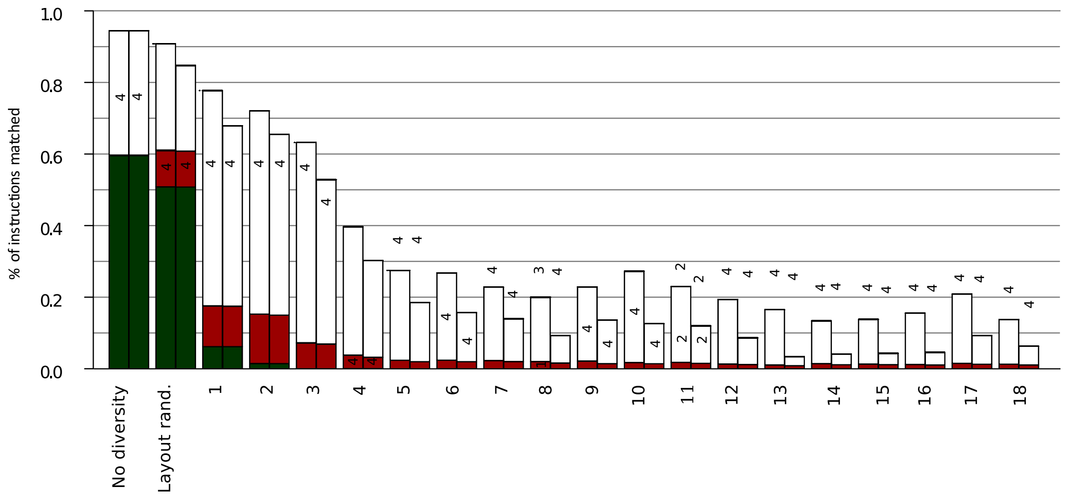


Fig. 11. Diffing results for the `pngtest_debian` use case according to match quality.

The leftmost "iteration" in both charts corresponds to diffing the unpatched binary with the undiversified patched binary. It is clear that in that case BinDiff is doing a really good job. As indicated in blue, many procedure matches are found by computing hashes over the ordered instructions in the procedures. The hashes neglect immediate operands to abstract away changed offsets and changed absolute addresses. Most of the matches are then found recursively on the CG through so called "call sequence" matchers: these build on the assumption that procedures of which the callers match are likely matches themselves.

The second iteration from the left in both charts is iteration zero, in which only code layout randomization is applied. It is clear that while BinDiff is hampered by this randomization, it still does a pretty good job. The matching is mainly based on more abstract CFG properties, however, that do rely less on the order and occurrence of individual instructions.

As soon as we start diversifying the code, the effectiveness of BinDiff starts to drop. Almost immediately, the "very good" quality matchers start to fail and the lower quality metrics take over. The matcher based on the strings that are referenced in procedures then becomes quite important, along with the different recursive matchers based on the CG. As more and more diversification is introduced during our iterative approach, the amount of matched code drops significantly, and the amount of correctly matched code drops even lower.

The reason why the amount of (correct) matches fluctuates in the later iterations is due to the random layout randomization. Every time a CFG is transformed, the layout, which is determined by successive invocations of the PRNG, changes globally. In some iterations it changes for the worse, sometimes for the better.

In this experiment, the best result is obtained after 14 iterations. For the binary generated in that iteration, BinDiff can match less than 5% of the code, and all relevant instructions are in the 95% unmatched code. So clearly BinDiff is of almost no use to an attacker at this point.

After 18 iterations, an even better results was obtained, but as we will see, that result was achieved at the expense of more performance overhead.

For completeness, figures 11, 12, 13 show the diffing results for the other three use cases. It is clear that also in these cases, our approach makes BinDiff almost completely useless to attackers.

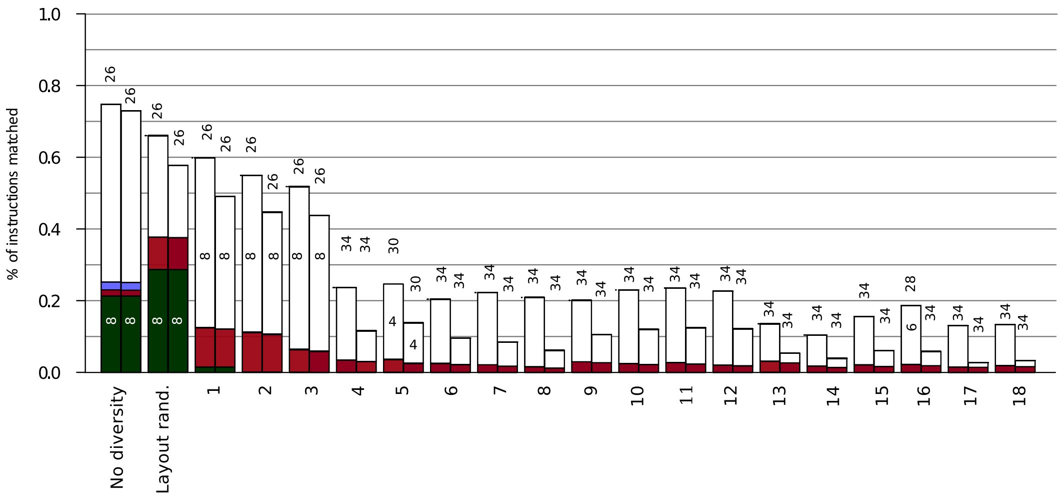


Fig. 12. Diffing results for the `pngtest_beta` use case according to match quality.

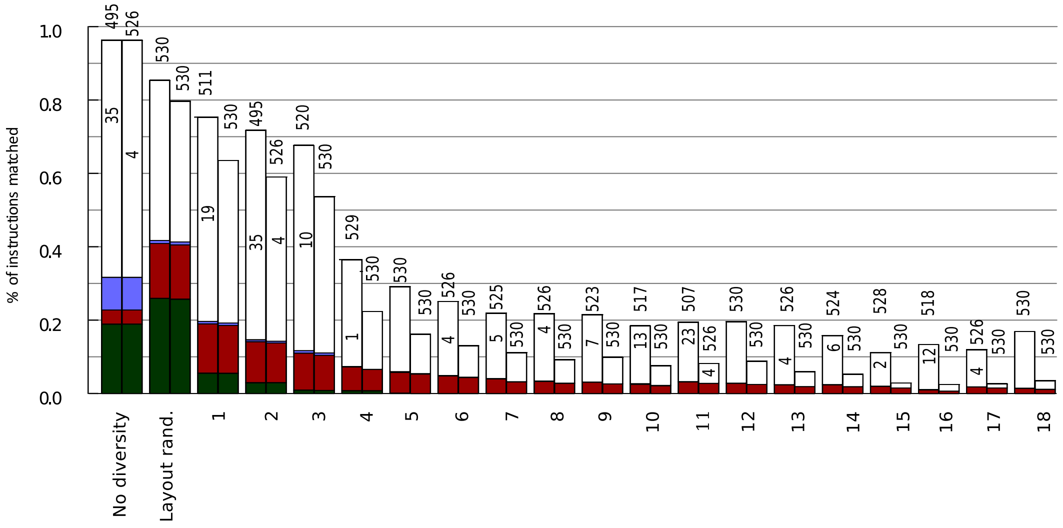


Fig. 13. Diffing results for the `soplex` use case according to match quality.

4.4. Overhead

Figure 14 depicts the overhead of the diversification in terms of code section size. Starting with layout randomization, every transformation adds some additional code. In iterations 7 and 13, when new transformations become applicable on executed fragments according to the rules in Table I, the most overhead is added. That is precisely why we wait so long for applying these transformations. For the most interesting versions of the binaries, the code size overhead is 15-25%. We believe this to be acceptable.

Figure 15 depicts the overhead of diversification in terms of binary patch size. The patches become significantly bigger, up to the point where their size becomes between 30% and 40% of the full code section size. For isolated patches, such as in the `pngtest_debian` use case, diversification can increase the binary patch size with a factor 1000. For bigger patches, as for `pngtest_beta`, the overhead is limited to a factor 2.5. The overhead to distribute diversified

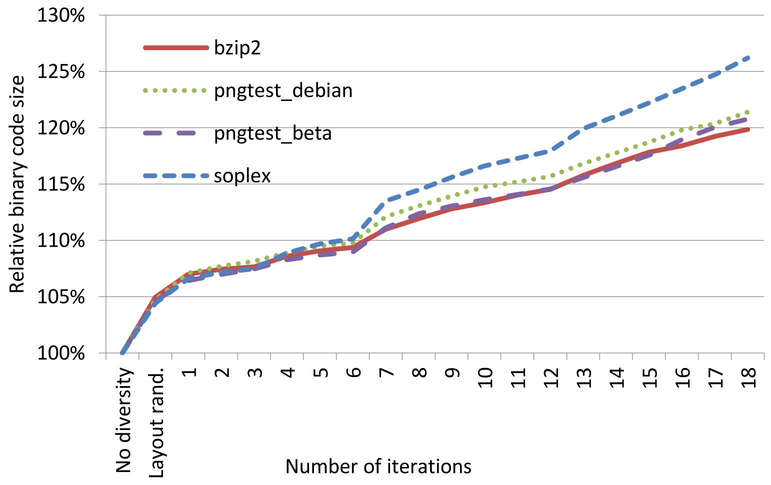


Fig. 14. Relative binary code sizes

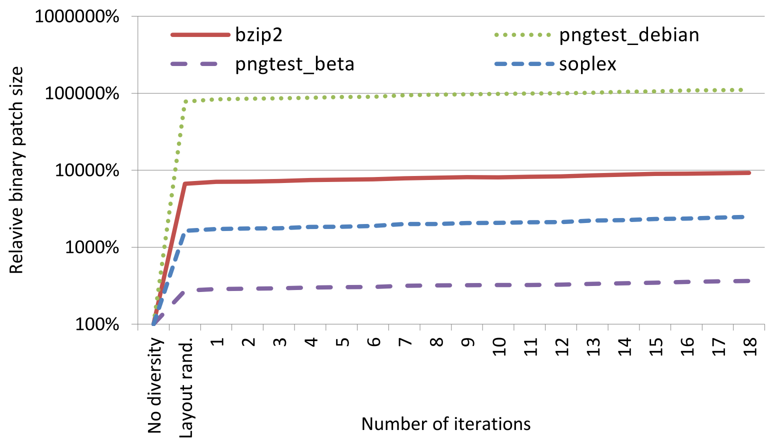


Fig. 15. Relative binary patch sizes

patches, e.g., over the Internet, is hence extremely variable. The trade-off between this overhead and the provided protection against patch-based attacks is one of the trade-offs that developers will have to make.

For the benchmarks for which we have training and reference inputs from the SPEC benchmark suite, Figure 16 presents the performance overhead of our approach.

For `bzip2`, we observe very low overheads until the last but two iterations. The overhead is even negligible for the first 9 iterations. For `soplex`, we obtain very low overhead as well, but it is significant from the very first iterations onwards, and it surpasses 5% as of iteration 13. At that iteration, BinDiff was only able to correctly match about 6% of all code.

So we can conclude that our approach is able to thwart BinDiff effectively and efficiently. Moreover, as the chart in Figure 17 shows, the developer can clearly trade-off protection vs. overhead.

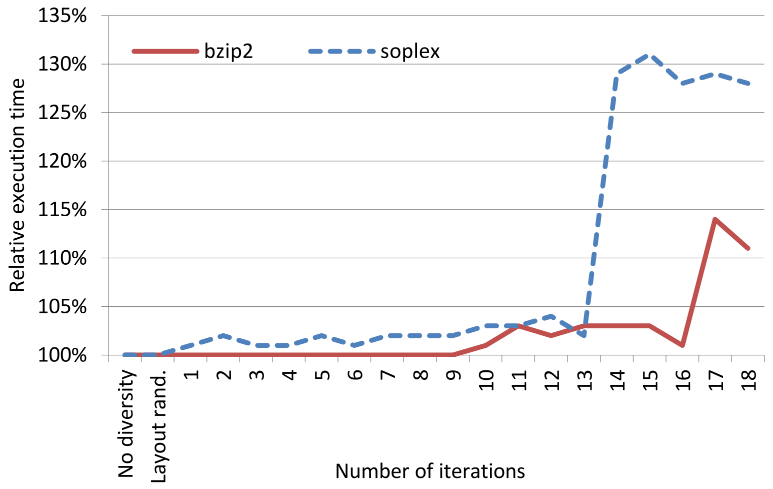


Fig. 16. Execution times relative to the times of the undiversified binaries.

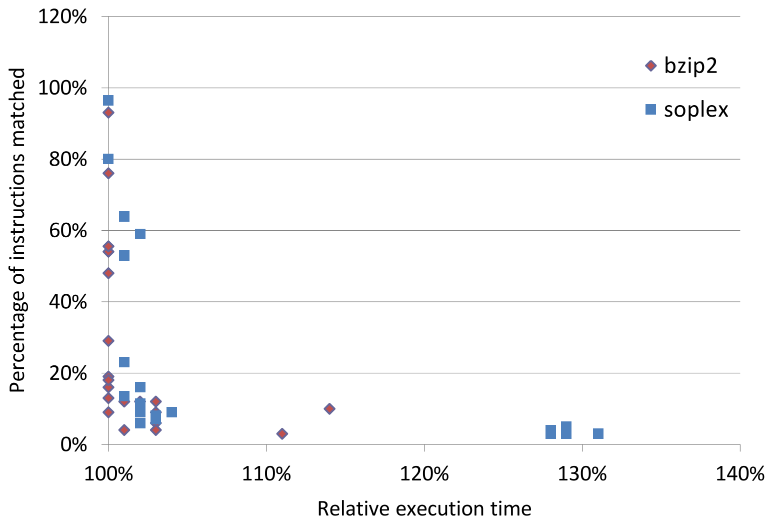


Fig. 17. Relative execution times vs. percentage of correctly matched instructions.

5. RELATED WORK

5.1. Software Matching

Matching two versions of some piece of information has many applications, ranging from text file comparison to DNA matching. In this section, we briefly discuss related work relevant to the matching of binary program versions.

An excellent overview of existing program matching techniques is available from Kim and Notkin [Kim and Notkin 2006]. However, the techniques they survey are all situated in the context of software development and software engineering, where program changes need to be tracked during the evolution of a program (branches). Such tracking may be required, e.g., to update regression test suites or to update profile information. That context is completely different from ours, and the techniques they survey are not applicable in our context of attack diversified programs. And neither is their own matching technique [Kim et al. 2007].

5.1.1. Text-based Matching Approaches. Text-based matching algorithms are about finding the minimum script of symbol deletions and insertions that transform one sequence into another. This type of matching is used in, e.g., spelling correction systems and file comparison tools [Miller and Myers 1985; Wagner and Fischer 1974]. However, due to the limitations on the types of changes (insertions and deletions), they cannot fully capture the degrees of freedom in the language of programs. While this typically is not a problem in the application domain of text-based matching algorithms, it does become problematic when, e.g., statements have been deliberately reordered.

5.1.2. Graph-based Matching Approaches. More recently, graph-based binary matching algorithms [Dullien and Rolles 2005; Flake 2004; Sabin 2004; Gao et al. 2008] have been proposed like the ones that are used in BinDiff. They compare high-level structures like control flow graphs, as opposed to source code, assembly or code bytes. Often this offers a more illuminating view of relevant differences between two program versions. This approach can foil diversification methods that do not alter the control flow graph much, e.g., reordering transformations. BMAT [Wang et al. 2000] is a another binary matching tool that has been developed with the primary goal of reusing profile information in subsequent builds.

As we demonstrated in this paper and as we observed in some experiments we ran with BinHunt with similar results, these approaches are easily thwarted by diversification techniques that alter the CFGs and CGs.

5.1.3. Trace-based Matching Approaches. Trace-based matching approaches collect information about the execution of the program, such as control flow, values produced, addresses referenced and data dependencies exercised. Techniques based on compact representations of dynamic program slices [Zhang and Gupta 2005b] have been evaluated by comparing unoptimized and optimized versions of a program [Zhang and Gupta 2005a].

Similar techniques have been used to compare original and obfuscated versions as well [Nagarajan et al. 2007]. Nagarajan et al. describe a technique that consists of two steps: an interprocedural matching step and an instruction matching step. The goal of the first step is to produce a mapping between the functions of two program versions. To enable this matching, each function is associated with a signature. By comparing signatures, *compatible* functions can be determined. The compatible functions are then matched using the structure of *dynamic call graphs* (DCGs) of the two executions, which essentially yields the function mapping. In the second step, an attempt is made to match the instructions within the matching functions. To enable this matching, each instruction is associated with a signature. By comparing signatures, compatible instructions can be determined. The compatible instructions are then matched using the structure of the *dynamic data dependence graphs* (DDDGs) of the two program versions. The DDDGs are matched using the iterative algorithm discussed in [Zhang and Gupta 2005a]. First, the root nodes of the DDDGs are matched by comparing the signatures. Then the interior nodes of the DDDGs are matched using an algorithm that iteratively applies two passes. In the forward pass, nodes, all of whose parent nodes match, are in turn matched. In the backward pass, nodes, at least one of whose children nodes match, are in turn matched. Repeated iteration of each of these passes, iteratively refines the instruction matches. Nagarajan et al. evaluated their technique by matching original program versions to obfuscated versions that were generated with the a previous version of the tool used later in this paper to generate diverse versions for this paper. Their results were very good, and they obtained extremely low false-positive and false-negative rates.

However, when that matching technique was applied to two diversified versions of a program, the results were extremely bad, even after attempts to tune the algorithm for this new application context. For example, after matching two diversified versions of `bzip2`, the tuned technique still reports more than 100000 instruction matches, even though the number of covered instructions in the original program version of bzip is only 13609. The 100000 matches result in a false-positive rate of 97%, and still there is a false-negative rate of 81%. As with BinDiff and other tools, the reason for this ineffectiveness is in the failing procedure signature matching. So also this technique will not likely be very useful to an attacker when software is diversified as we proposed.

5.1.4. Polymorphic Malware Analysis. One specific context in which software matching techniques is playing an increasingly important role is the analysis of polymorphic malware [Anderson et al. 2011]. The malware tries to avoid being detected by mutating, Anti-malware tools try to abstract the mutations. Reasons for this abstraction include being able to collect and combine information from multiple versions of the malware [LeDoux et al. 2012] and faster classification of malware samples [Bayer et al. 2010].

One specific method to abstract mutations between different samples is to normalize them before analyzing and matching them [Walenstein et al. 2008]. It is an open research question to what extent such normalization could damage the effectiveness of our approach. We know of no publicly available prototype tools that implements normalization.

5.2. Diversification

To thwart diffing and matching tools, we will rely on software diversification techniques. To defend *against malicious code* attacks, software diversification (a.k.a. individualization) was first proposed by Cohen [Cohen 1993] under the term “program evolution”. Since, numerous transformation techniques have been presented, including control flow transformations [Anckaert 2008], memory layout randomization [Bhatkar et al. 2003; Forrest et al. 1997] and randomizing the instruction set [Barrantes et al. 2005; Kc et al. 2003]. It has been shown that these techniques are vulnerable to attacks as well [Shacham et al. 2004; Sovarel et al. 2005]. Other research assumes the presence of diversity and studies the assignment of distinct software packages to individual systems in a network [O’Donnell and Sethu 2004] or uses different versions in a framework for detection and disruption of attacks [Cox et al. 2006] similar to N-version programming for fault tolerance [Avizienis and Chen 1977]. Software diversity as a protection mechanism *against a malicious host*, in which case some sensitive software is run on a host computer to which an attacker has full access, seems to have received less attention. Existing work is focused on randomization before distribution. Anckaert et al. [Anckaert et al. 2006] propose to rewrite the program in a custom instruction set and to ship it with a matching virtual machine. Zhou et al. [Zhou and Main 2006] present code transformations based upon algebraic structures compatible with 32-bit operations commonly present in code.

5.3. Instruction Set Limitation

Diffing tools can also be thwarted by making code more similar instead of more dissimilar. In particular, when multiple code fragments within a single program version are made more similar, diffing tools have a harder time differentiating between them. For example, all signatures computed on opcodes occurring in a procedure would fail on a single-instruction computer [Jones 1988].

A more practical demonstration of this approach was presented by De Sutter et al. [De Sutter et al. 2008]. They replace infrequently occurring x86 instructions, which provide matching hooks for matching tools, by sequences involving more frequently instructions. With that approach, the false matching rates of a trace-based matcher that involved instruction opcode signatures but also CFG and DDG properties, went up by several tens of percentages.

6. DISCUSSION

The results from Section 4 indicate that strong protection against patch attacks with BinDiff can be obtained at a negligible performance overhead, but at a significant cost in patch size. Several issues can still be raised about the proposed approach, however.

First, one can question whether the proposed approach is specific for BinDiff or whether it is more general. Our experiments with other IDA Pro diffing plug-ins, including the results presented earlier [Coppens et al. 2012], demonstrate that the approach is equally effective against all other publicly available plug-ins. However, it is always possible to extend the attack tools with new analyses and transformations, which could improve the success rate of an attacker. Attackers could try to detect and undo specific transformations, including the obfuscations applied in our approach. Deobfuscating transformations are designed to return simplified binaries, however, not binaries that can be more easily compared with other binaries. So deobfuscating transformations do not necessarily return a unique representation when given two differently obfuscated but semantically equivalent binaries. Moreover, some of the most effective attacks like dynamic or hybrid static-dynamic deobfuscation [Sharif et al. 2009; Madou et al. 2005] cannot be used to exploit patches. In such dynamic attacks, the program's execution is first monitored and traced. The traces are then used to remove some of the never-taken execution paths from the program. This builds on the assumption that never-taken execution paths with certain signatures probably originate from obfuscating transformations such as opaque predicates, and hence were not present in the original program. In the case of patches that insert input validation checks that the attacker cannot trigger yet, the traces he collects on the patched program will indicate that the inserted checks are potential opaque predicates. So instead of helping the attacker to obtain a better diffing result, his deobfuscation will hide the patch. Against more advanced tools such as deofbuscators and code normalization tools, the effectiveness of our approach remains to be studied. However, this situation is not uncommon in the software protection arms race: once a software protection scheme is in use, attackers will try to break the transformations used. However, the fact that particular transformations are defeated does not imply that the whole approach using those diversifying transformations is flawed or broken. When attacker tools become more effective, we can extend the set of relatively simple transformations in our current implementation with more complex ones to make attackers require even more complex tools. While the complexity of our current set of link-time transformations is limited because of the lack of high-level semantic information in object files, similar as well as much more complex diversifying transformations can easily be integrated into a compiler. In general, the more complex the tools in the attacker tool box need to become to overcome the protection provided by diversification, the more time-consuming they will be, and hence the smaller the attacker's window of opportunity will become. So we are quite confident that our approach, although it might have to be tuned and extended in the future, provides a solid foundation for protecting against patch-based attacks.

Secondly, it is worth mentioning that our approach can easily be extended to protect successive patch releases. When a patch to v3 is released, and protection against collusion attacks against both v1 and v2 is required, it suffices to use a new set of PRNG seeds and to run the diffing tool twice in each iteration to diff v3 against v1 and against v2, and to consider the union of the sets of matched procedures in the next iteration.

Third, it is important to discuss some more qualitative, less quantitative types of costs of our approach. In particular, we have to look at the impact our approach has on the customer support and code maintenance cost. This depends on the ease with which one can debug the code and interpret bug and crash reports. In this regard, we should point out that our approach so far only involves control flow transformations. The original code is not rescheduled, register allocation is not changed, and all data layout remains untouched. The latter includes the statically allocated data, as well as the stack (frames) and the heap.

As the developers know the mapping between code fragments in the non-diversified binary and in the diversified binary, and all data addresses remain unchanged, we conjecture that debugging the diversified binaries or interpreting crash reports of them is not significantly harder.

With regard to both qualitative and quantitative costs of using our approach, one also has to consider the trade-off between protection and overhead that can be made on a case by case basis. Whether or not the protection is worth the overhead will, e.g., depend on the criticality of a patch. When a white-hat hacker contacts a developer about a zero-day vulnerability for which he has an exploit, 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 and no exploit has ever been constructed, there will be little need to protect a patch.

Finally, we should mention that our approach does not rely on security through obscurity. What we try to protect against is the identification of the semantic changes resulting from a binary code patch. We do so by engineering a secret rule set and by using secret PRNG seeds. But the approach itself and the tools used can be public.

Once enough users have applied the patch, there is no longer a need for the details of the semantic changes to remain protected. It is then up to the software vendor to make a trade-off between the time the semantic changes are protected and how much time users have to apply the patch. In fact, a similar trade-off exists in the open source community, where it can be argued that security patches should be developed into a private repository and only put into a public repository at a later time to hide the semantics of security patches before they are released to the public [Barth et al. 2011].

7. CONCLUSIONS AND FUTURE WORK

We have presented an iterative, feedback-driven compiler tool flow to diversify patched software with the goal of preventing the identification of patched code fragments by attackers. We have demonstrated that the approach is able to render the most commonly used diffing tool, BinDiff, and similar tools almost completely useless for attackers, at minimal or even negligible performance overhead.

Our approach significantly increases the manual investigation effort required to mount exploits against patched vulnerabilities, and thus shortens the window of opportunity for patch-based exploits.

As future work, we consider fine-tuning the rule sets to make the approach even more efficient and effective. The currently used set was engineered manually, we plan to make this an auto-tuning feature by means of machine learning. Furthermore, we plan to include more types of diversifying transformations and to evaluate our approach against trace-based diffing tools.

REFERENCES

Anckaert, B. 2008. Diversity for software protection. Ph.D. thesis, Ghent University.

Anckaert, B., Jakubowski, M., and Venkatesan, R. 2006. Proteus: virtualization for diversified tamper-resistance. In *Proceedings of the workshop on Digital Rights Management.* 47–58.

Anderson, B., Quist, D., Neil, J., Storlie, C., and Lane, T. 2011. Graph-based malware detection using dynamic analysis. *Journal in Computer Virology 7*, 247–258. 10.1007/s11416-011-0152-x.

Avizienis, A. and Chen, L. 1977. On the implementation of N-version programming for software fault tolerance during execution. In *The 1st IEEE Computer Software and Applications Conference.* 149–155.

Barrantes, E. G., Ackley, D., Forrest, S., and Stefanovi, D. 2005. Randomized instruction set emulation. *ACM Trans. on Info. and Syst. Secu. 8,* 1, 3–40.

Barth, A., Li, S., Rubinstein, B., and Song, D. 2011. How open should open source be? arXiv:1109.0507v1.

Barthen. 2009. [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`.

Bayer, U., Kirda, E., and Kruegel, C. 2010. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. ACM, New York, NY, USA, 1871–1878.

Bhatkar, S., DuVarney, D., and Sekar, R. 2003. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *The 12th USENIX Security Symposium*. 105–120.

Boneh, D. and Shaw, J. 1998. Collusion-secure fingerprinting for digital data. *IEEE Transactions on Information Theory 44,* 5, 1897–1905.

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

Cohen, F. 1993. Operating system evolution through program evolution. *Computers and Security 12,* 6, 565–584.

Collberg, C., Thomborson, C., and Low, D. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th Conference on Principles of Programming Languages*. ACM Press, 184–196.

Coppens, B., De Sutter, B., and De Bosschere, K. 2012. Protecting your software releases. *IEEE Security & Privacy*. Accepted for publication on 5 September 2012.

Core Security Technologies. 2010. Windows SMTP service DNS query id vulnerabilities. CoreLabs Security Advisory.

Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., and Hiser, J. 2006. N-variant systems: A secretless framework for security through diversity. In *The 15th USENIX Security Symposium*. 105–120.

De Sutter, B., Anckaert, B., Geiregat, J., Chanet, D., and De Bosschere, K. 2008. Instruction set limitation in support of software diversity. In *11th International Conference on Information Security and Cryptology ICISC 2008*. Number 5461 in LNCS. 152–165.

Dullien, T. and Rolles, R. 2005. Graph-based comparison of executable objects. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications*.

Eagle, C. 2011. *The IDA Pro Book* 2nd Ed. No Starch Press.

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

Ergun, F., Kilian, J., and Kumar, R. 1999. A note on the limits of collusion-resistant watermarks. *1592*, 140–149.

Flake, H. 2004. Structural comparison of executable objects. In *Proceedings of the Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop*. 161–173.

Forrest, S., Somayaji, A., and Ackley, D. 1997. Building diverse computer systems. In *The Workshop on Hot Topics in Operating Systems*. 67–72.

Frijters, J. 2010. Reverse engineering the MS10-060 .NET security patch. Blogpost.

Gao, D., Reiter, M. K., and Song, D. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 10th International Conference on Information and Communications Security*. ICICS '08. Springer-Verlag, Berlin, Heidelberg, 238–255.

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

Johnson, N. 2011. From patch to proof-of-concept: MS10-081. Blogpost.

Jones, D. W. 1988. The ultimate risc. *SIGARCH Comput. Archit. News 16,* 3, 48–55.

Kc, G., Keromytis, A., and Prevelakis, V. 2003. Countering code-injection attacks with instruction-set randomization. In *The 10th ACM Conference on Computer and Communications Security*. 272–280.

Kim, M. and Notkin, D. 2006. Program element matching for multi-version program analyses. In *Proceedings of the 2006 international workshop on Mining software repositories*. 58–64.

Kim, M., Notkin, D., and Grossman, D. 2007. Automatic inference of structural changes for matching across program versions. In *Proceedings of the 29th international conference on Software Engineering*.

LeDoux, C., Walenstein, A., and Lakhotia, A. 2012. Improved malware classification through sensor fusion using disjoint union. In *Proceedings of the 6th International Conference on Information Systems, Technology and Management (ICISTM)*. 360–371.

Lee, B. and Jang, Y. 2012. Exploit shop website.

Linn, C. and Debray, S. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. ACM Conf. on Computer and Communications Security*. 290–299.

Loveless, M. 2006. Corporate security: A hacker perspective.

Madou, M., Anckaert, B., De Sutter, B., and De Bosschere, K. 2005. Hybrid static-dynamic attacks against software protection mechanisms. In *Proceedings of the 5th ACM workshop on Digital Rights Management.* 75–82.

Miller, W. and Myers, E. 1985. A file comparison program. *Software - Practice & Experience 15,* 11, 1025–1040.

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

Nagarajan, V., Zhang, X., Gupta, R., Madou, M., De Sutter, B., and De Bosschere, K. 2007. Matching control flow of program versions. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance.* 83–94.

O'Donnell, A. and Sethu, H. 2004. On achieving software diversity for improved network security using distributed coloring algorithms. In *Proceedings of the 11th ACM conference on Computer and Communications Security.* ACM Press, 121–131.

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

Percival, C. 2003. Naive differences of executable code. `http://www.daemonology.net/bsdiff/`.

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

Sabin, T. 2004. Comparing binaries with graph isomorphisms. Tech. rep., BindView RAZOR Team.

Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security.* ACM Press, 298–307.

Sharif, M., Lanzi, A., Griffin, J., and Lee, W. 2009. Automatic reverse engineering of malware emulators. In *IEEE Symposium on Security and Privacy.*

Slawlerguy. 2008. Reversing the ms08-067 patch... Blogpost.

Sotirov, A. 2006. Reverse engineering Microsoft binaries. CanSecWest.

Sovarel, A., Evans, D., and Paul, N. 2005. Where is the FEEB? The effectiveness of instruction set randomization. In *Proceedings of the 14th USENIX Security Symposium.* 145–160.

Varghese, N. 2008. Reverse engineering for exploit writers. Clubhack.

Wagner, R. and Fischer, M. 1974. The string-to-string correction problem. *Journal of the ACM 21,* 1, 168–173.

Walenstein, A., Mathur, R., Chouchane, M. R., and Lakhotia, A. 2008. Constructing malware normalizers using term rewriting. *Journal in Computer Virology 4,* 4, 307–322.

Walia, H. 2011. Reversing Microsoft patches to reveal vulnerable code. Nullcon.

Wang, C., Davidson, J., Hill, J., and Knight, J. 2001. Protection of software-based survivability mechanisms. In *Proceedings of the 2nd International Conference of Dependable Systems and Networks*. IEEE Computer Society Press, 193–202.

Wang, Z., Pierce, K., and McFarling, S. 2000. Bmat – a binary matching tools for stale profile propagation. *The Journal of Instruction-Level Parallelism 2*, 1–20.

Zhang, X. and Gupta, R. 2005a. Matching execution histories of program versions. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering.* 197–206.

Zhang, X. and Gupta, R. 2005b. Whole execution traces and their applications. *ACM Trans. on Architecture and Code Optimization 2,* 3, 301–334.

Zhou, Y. and Main, A. 2006. Diversity via code transformations: A solution for NGNA renewable security. In *NCTA - The National Show.*

zynamics. 2012. BinDiff. `http://www.zynamics.com/bindiff.html`.

Zynamics 2012. *Zynamics BinDiff Manual.* Zynamics.