Ghost Patches: Fake Patches for Fake Vulnerabilities

Jeffrey Avery(✉) and Eugene H. Spafford

Computer Science Department and CERIAS,
Purdue University, 305 N. University St., West Lafayette, IN 47907, USA
{avery0,spaf}@purdue.edu
https://www.cs.purdue.edu/

Abstract. Offensive and defensive players in the cyber security sphere constantly react to either party's actions. This reactive approach works well for attackers but can be devastating for defenders. This approach also models the software security patching lifecycle. Patches fix security flaws, but when deployed, can be used to develop malicious exploits.

To make exploit generation using patches more resource intensive, we propose inserting deception into software security patches. These *ghost patches* mislead attackers with deception and fix legitimate flaws in code. An adversary using ghost patches to develop exploits will be forced to use additional resources. We implement a proof of concept for ghost patches and evaluate their impact on program analysis and runtime. We find that these patches have a statistically significant impact on dynamic analysis runtime, increasing time to analyze by a factor of up to 14x, but do not have a statistically significant impact on program runtime.

1 Introduction

Software developers release programs to the public every day, but this code is not perfectly written. As stated by Mosher's Law of Software Engineering: *"Don't worry if it doesn't work right. If everything did, you'd be out of a job"* [18]. Thus developers release code that has flaws and subsequently provide the necessary patches to fix the flawed code.

These patches are released with varying frequency, depending on the severity of the flaw as well as developer resources. Software security patches have a higher severity rating than non-security patches because these vulnerabilities could negatively impact other programs and services present on the machine. Thus, when a security flaw is discovered, a patch to fix this flaw usually follows shortly after. This trend is shown by a report released in 2016 where the average time the top five most frequently exploited zero-day vulnerabilities remained undetected once an attack was released was 7 days and the average time to install a patch for these vulnerabilities once the patch was released was 1 day [23].

Despite the speed with which these flaws are detected and updated, patches also inherently have a negative impact on the software: revealing the location of

S. De Capitani di Vimercati and F. Martinelli (Eds.): SEC 2017, IFIP AICT 502, pp. 399–412, 2017.
DOI: 10.1007/978-3-319-58469-0_27

vulnerabilities and the type of security vulnerability being patched. This provides a blueprint for attackers as they develop exploits for vulnerable systems. Exploit generation research shows that attackers can use patches to find vulnerabilities and use these results to develop exploits for unpatched code [6,10].

One solution to make this blueprint more difficult to understand is to obfuscate the traditional patch [4,9,25]. This approach will make the patch more difficult to statically analyze, but dynamic analysis tools exist that can analyze obfuscated code [24] and the location of the patch is not masked. Therefore, additional techniques must be applied to software patching to enhance its security.

Thus, an additional technique we propose is to apply deception to software security patching to enhance its security. One method we propose to deceive attackers injects a faux patch, composed of one or more fake patches into traditional patches. These misleading updates could influence attackers' ability to identify a legitimate patch, making exploit generation using patches more resource intensive. While not an absolute solution to the problem of patch-based exploit generation, these misleading updates could increase the resources needed to reverse-engineer patches, which is the first step in exploit generation and along with other deceptive and detection techniques could make these attacks more cost intensive. We call the combination of a faux patch and a legitimate patch(es) for a single program a *ghost patch*.

Definition. Ghost patches are composed of two components: a legitimate patch, made of traditional patches, and a faux patch, made of fake patches. The legitimate patch component fixes any vulnerability or vulnerabilities that may exist in the program. These traditional patches are actual fixes for vulnerable code. The faux patch component is additional code meant to mislead attackers. Each fake patch within a faux patch suggests a fake vulnerability.

Ghost patches provide the same level of security as traditional patches, but could confuse attackers analyzing the code. This extra time and effort attackers spend analyzing the code would increase the time between a patch release and an exploit release. This could provide end-users more time to patch their vulnerable systems, causing fewer attacks to succeed.

Related Work. Researchers have applied deceptive techniques to software patches, but we believe our work is the first full treatment that applies, implements and analyzes fake code to software security patches, specifically patches for input validation vulnerabilities. Araujo et al. [2,3] apply deception to security vulnerabilities such as Heartbleed and Conflicker. Their work focuses on fixing a patch by detecting an attack and diverting the runtime environment to a sanitized virtual environment that appears vulnerable. Researchers though have developed techniques to reliably distinguish sandbox environments from real user machines [27]. Thus, an attacker can identify when they are being monitored and deceived. This technique also does not camouflage legitimate patches, thus, an adversary can easily identify these patches using a diff between a patched and unpatched program. Ghost patches do mask the location of legitimate patches among other plausible code updates, making an adversary's task of identifying the legitimate patch more resource intensive.

Fake patches have also been mentioned by Bashar et al. [4] and Oh [19] as ways to deceive attackers. Our work differs from these by implementing a compiler solution using LLVM [15] to add fake patches to code and evaluating their impact on program analysis and runtime. Adding false code to improve code stealth and make reverse engineering of an entire program more difficult has been explored [14].

Work by Collberg et al. also discuss *bogus control flow* statements [8]. Bogus control flow statements are control flow statements that mislead reverse engineering techniques and static analysis tools. This technique focuses on making legitimate paths in a program more difficult to identify to prevent attackers from bypassing sections of code (i.e. registration, validation, DRM, etc.). This work is similar to our approach, as both attempt to add additional statements making code more difficult to understand. Bogus control flow mainly attempts to prevent bypassing critical code segments, while ghost patches attempt to increase the resources necessary to develop patch-based exploits. Our approach differs as dynamic and static analysis tools will be influenced by our approach (i.e. increase analysis runtime) while applying dynamic analysis to bogus control flows will expose the legitimate path in the code. Researchers also have presented the idea of inserting beaconing code traps where return-oriented-programming (ROP) gadgets would be expected by adversaries [11]. Our work differs as we implement an automated technique to enhance the security of patches for input validation vulnerabilities. Coppens et al. suggest code diversification can be introduced using patches to increase the effort necessary for exploit generation [10]. This approach would force attackers to generate multiple exploits to achieve widespread compromise, increasing the effort and resources needed to exploit a program compared to everyone having the same code with the same vulnerability. This work though does not attempt to mask the legitimate patch, thus an attacker could easily identify and analyze legitimate patches to code that fix legitimate vulnerabilities.

Our contributions from this work are as follows:

- Presentation of a novel methodology to develop and insert fake patches.
- Implementation of a proof-of-concept to inject fake patches in code with input validation vulnerabilities that could result in integer over/under-flows.
- Experimentation with dynamic analysis tool to analyze impact of fake patches.

The rest of this work is organized as follows: Section 2 discusses background information, Sect. 3 introduces our approach to applying deception to put validation patches, Sect. 4 explains how we evaluate a faux patch, Sect. 5 presents results from evaluating our proof of concept, Sect. 6 reviews limitations and challenges of ghost patches as well as discusses potential solutions, and Sect. 7 concludes this work.

2 Background

Deception. Deception has been used in computing since the 1970s [12,16,21,22]. Since its introduction, a variety of deceptive tools have been developed to bolster computer defenses. Examples of deceptive tools are those that generate decoy documents [20] and honeyfiles [29]. These documents are planted to attract attention away from critical data or resources and alert defenders of potential intrusions or exfiltration attempts.

Our research will use decoy document properties to develop fake patches. The definition of deception in computation states that any intentional act of misleading to influence decision making classifies an act as deception [1,28]. Thus, *ghost patches* are intentionally placed in code to mislead attackers. Deception can also be broken into two components, simulation, showing the false, and dissimulation, hiding the real. Simulation and dissimulation are broken down into three separate components. Simulation is comprised of mimicking, inventing, and decoying and dissimulation is comprised of masking, repackaging, and dazzling [26].

Fake patches are an application of showing the false by *mimicking* and *decoying* and hiding the real by *dazzling*. They show the false by including characteristics of real patches, mimicking a real patch and attracting attention away from traditional patches as a decoy. Fake patches hide the real by reducing the certainty of which patches are real and which are decoys.

Exploiting Patches. Attackers can use patches to develop exploits. One approach statically reverse-engineers the patched code to determine the vulnerability being fixed. Another approach dynamically analyzes patched code to determine new paths that have been added compared to unpatched code. These new paths suggest that a vulnerability can be exploited in the unpatched program by generating input that follows the new path in the patched program.

Using either approach, attackers can view the actual lines of code being changed among program versions. With our approach, fake patches will be presented along with traditional patches, forcing adversaries using static analysis to distinguish between each type of patch before generating malicious input. The fake patches that we add can be executed by benign input without altering the program's semantics, forcing adversaries using dynamic analysis to distinguish which paths are legitimate and which are deceptive. Research has also shown that exploits can be generated automatically and quickly based on detecting new "checks" in patched code [6].

Input Validation Vulnerabilities. This work targets input validation vulnerabilities. A common patch to these types of vulnerabilities is to add boundary checks in the form of if-statements [6]. Thus, given a patch and unpatched program, a diff between the two programs will show additional branch statements in the patched version. These branch statements can be used to then determine input values that will exploit an unpatched program.

3 Approach

This research studies how a fake patch can be implemented in conjunction with a traditional patch and measures its impact on program analysis and runtime.

These fake patches should alter the control flow of a program, but not the data flow of information. Thus, given two programs, one with a ghost patch and the other with a traditional patch, the final output should be identical.

Our approach is based on the trend that input validation vulnerabilities are patched by adding conditional statements that validate the value of variables that can be tainted by malicious input [6]. Thus, to deceive attackers, we add fake patches to code that mimic these input validation conditional statements, making exploit generation using patches more resource intensive.

3.1 Threat Model

We consider attackers who are using patches to develop exploits and have access to both patched and unpatched versions of a program, and can control and monitor the execution of both as our threat model.

Ghost patching is designed for input validation vulnerabilities that have not been discovered by the public or do not have a widely available exploit. If there are scripts that already exploit a well known vulnerability, ghost patches can still be applied but with less effectiveness. Public exploit databases[1] or "underground" forums could be monitored to determine if exploits have been developed.

We specifically look at input validation vulnerabilities that involve integers. These vulnerabilities can be exploited because of a lack of boundary checking and can cause subtle program misbehavior through integer overflows or underflows.

Finally, ghost patches target input validation vulnerabilities in enterprise scale systems. Due to performance constraints, embedded or real time systems do not present a suitable environment for ghost patches.

3.2 Properties of Ghost Patches

This work applies concepts from decoy documents to deceptive patches. Decoy documents are fake documents inserted into a file system or on a personal computer and are meant to intentionally mislead attackers. These documents also *mimic* real documents and are *decoys* meant to attract attention away from critical data. Bowen et al. and Stolfo et al. have conducted research on decoy documents [5,20] and created a list of properties that decoy documents should embody. We slightly modify these properties and present in Table 1 our list of fake patch properties as well as whether the property is trivial to implement or requires further experimentation.

3.3 Implementation Properties

The implementation of fake patches applies deception to patching because it attracts attention away from a traditional patch, but does not impact the data flow of the function being patched. Fake patches should be designed such that they are not marked as *dead-code* and removed from the binary as a result of

[1] https://www.exploit-db.com/.

Table 1. Fake patch properties

Property	Explanation	Implementation effort
Non-interfering	Fake patches should not interfere with program output nor inhibit performance beyond some threshold determined on a case to case basis	Experimentation
Conspicuous	Fake patches should be "easy" to locate by potential attackers	Easy
Believable	Fake patches should be plausible and not immediately detected as deceptive	Easy
Differentiable	Traditional and fake patches should be distinguishable by developers	Experimentation
Variability	Fake patches should incorporate some aspect of randomness when implemented	Easy
Enticing	Fake patches should be attractive to potential attackers such that they are not automatically discarded	Experimentation
Shelf-life	Fake patches should have a period of time before they are discovered	Experimentation

compiler optimization nor should they be trivial to identify by attackers. These patches should also address the properties outlined in Sect. 3.2. Implementation components of a fake patch should at a minimum include at least one randomly generated value and a conditional statement. Other implementation specifics depend on the actual program being patched.

Control Flow. Fake patches having conditional statements that alter control flow will make them apparent to attackers using static and dynamic analysis tools. This addresses the *conspicuous* property. This also mimics the trend of patches for input validation vulnerabilities.

Mimicking this trend could deceive attackers by showing changes that are expected but fake, addressing the *enticing* property. Experimentation will show how fake patches effect overall program runtime, addressing the *non-interfering* property. We implement fake patch conditional statements such that they include the destination or left-hand-side of an LLVM intermediate representation *store* instruction in the original program mathematically compared to a randomly generated value. The use of a random value address the *variability* property.

We form the body of if-statements by adding code that solves different mathematical expressions with the original program's value as input. These expressions do not alter the value of the legitimate variable; thus, data flow is preserved. The body of fake patch statements should be plausible for the program being patched. This suggests that the body of a fake patch should be developed based on the behavior of the program being patched.

3.4 Post Testing

After applying a ghost patch to software, further testing should be conducted for the following:

1. Evaluating ghost patch impact on software runtime and program memory (i.e. lines of code).
2. Verifying ghost patch does not introduce incompatibilities by applying unit testing.

A ghost patch should be evaluated for its impact on the program's performance to determine if it is feasible. This determination is dependant upon each program and the execution environment of the program. The memory impact of a ghost patch should also be considered. The size of a ghost patch should be reasonable for end-users to download and apply to vulnerable systems. Developers should establish an upper threshold such that the feasibility is measurable and can be validated. Conjectures about patch size and acceptable runtime are outside of the scope of this research. We do analyze the statistical impact of ghost patches on program runtime and program analysis.

3.5 LLVM Workflow

The workflow of our LLVM prototype begins with a traditionally patched file (we assume developers have previously created a traditional patch). First, this traditionally patched file is compiled using *clang*. This creates intermediate representation bytecode of the traditionally patched program. Next, this file is compiled a second time, applying our ghost patch LLVM pass. This pass adds one or more fake patches to the traditionally patched file. The fake patches are also implemented in intermediate representation bytecode. This stage creates a new ghost patched program. Next, this ghost patched program is compiled into binary using the *clang* compiler. If the file being patched is part of a larger project, the build tool for the project should be mapped to clang to ensure the project gets compiled with the correct flag(s). After the ghost patched code is compiled, the patched and unpatched (this file is before any traditional patch has been applied) binaries are supplied to a binary diff tool, such as *bsdiff*, to create a patch file that can be distributed and applied to unpatched programs. A work flow diagram of this process is shown in Fig. 1.

3.6 Implementation

We implemented a proof of concept that addresses input validation vulnerabilities involving integer variables. We believe our approach can be extended to other variable types and data structures without loss of generality. Our implementation uses LLVM and is about 300 lines of C++ code.

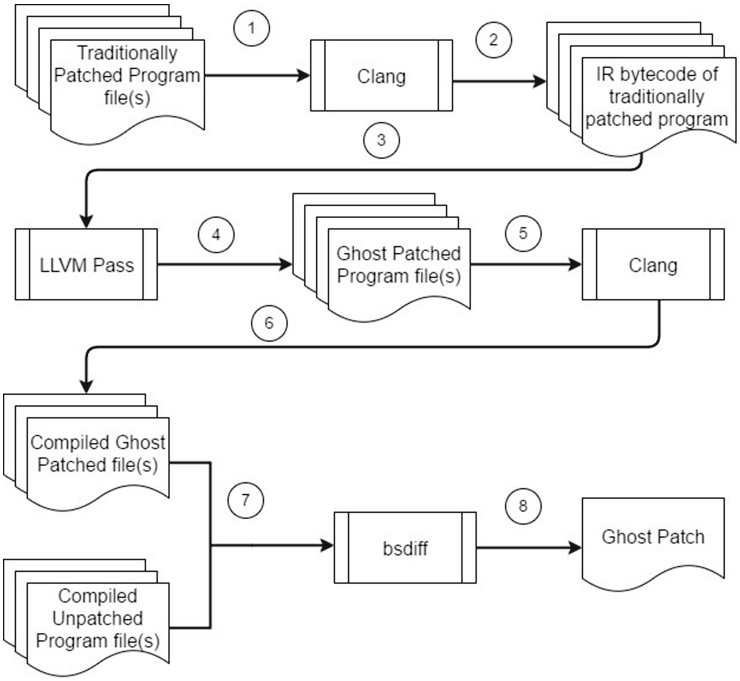
Traditionally Patched Program file(s)
1
Clang
2
IR bytecode of traditionally patched program
3
LLVM Pass
4
Ghost Patched Program file(s)
5
Clang
6
Compiled Ghost Patched file(s)
7
bsdiff
8
Ghost Patch
Compiled Unpatched Program file(s)

Fig. 1. Complete flow to create a ghost patch using LLVM and bsdiff

4 Evaluation

The prototype of an LLVM program was developed on an Ubuntu 14.04 x86_64 virtual machine. We used LLVM (version 3.4) to develop our pass because its front end compiler allows optimizations to be developed that can be applied to programs agnostic of the language they are written in. We use KLEE [7] (version 1.3) to dynamically analyze our vulnerable code because it evaluates paths through a program using symbolic execution, which can efficiently analyze programs without enumerating every possible input value. Thus, results from Klee represent a best case scenario for attacker resource utilization. The VM has 2 cores and 4 GB RAM. All experiments were also run on this virtual machine.

4.1 Simple Example

We evaluated our approach using the example below, which allows a user to enter two values and then copies each value into an integer variable and lacks input validation code. Then some, operations are performed and the results returned.

```
int calculate(int alpha, int beta);

int main(){
        int a,b,c;
        int d = 9;

        printf("Enter a value: \n");
        scanf("%d", &a);
        printf("Enter another value: \n");
        scanf("%d", &b);

        c = calculate(a,b);
        printf("Value of C: %d\n",c);

        a = b + d;
        if(a > 27)
                c = c * d;
        else
                b = a - b;

        d += d;
        return a;
}

int calculate (int alpha, int beta){
        if(alpha > 88)
                return (alpha + beta);
        else
                return (alpha * beta);
}
```

Experimentation. To evaluate our approach, we compare the length of time for Klee [7], a symbolic execution, dynamic analysis tool, to analyze a legitimately patched and faux patched version of the code. We use the runtime of Klee to measure the impact of a faux patch on exploit generation. We exploit the fact that each new branch will be analyzed because fake patches are indistinguishable from traditional patches from a software perspective.

To show the effect of our approach on program analysis, we evaluate whether the time to dynamically analyze traditionally patched code is significantly different statistically when compared to dynamically analyzing fake patched code using a t test. We also evaluated program runtime using this same experimental structure to determine fake patch's effect on program performance.

5 Results

5.1 Runtime Analysis

Using our simple code example, we collected runtime values using the *time* command for both the original program and a faux patched program. Figure 2 shows the difference in program runtime between a fake patched program and the unpatched program across 100 executions. Using this data, we determined the statistical significance of this difference in runtime using a *t* test. We concluded that there was no statistical significance between the runtimes for the original program and the faux patched program.

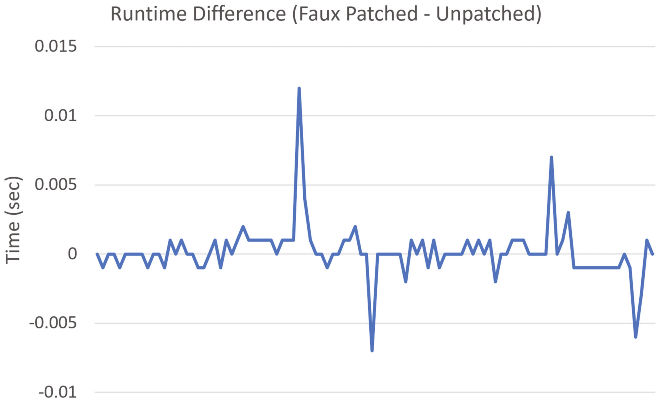

Fig. 2. Difference in faux patched vs. unpatched program runtime

5.2 Program Analysis

We collected values for the runtime of Klee using the *time* command as it analyzed an unpatched, traditionally patched and faux patched version of our simple code example. Figure 3 represents the runtime for each program across 100 executions. A *t* test using these values reveled that there is a statistical significance in Klee's runtime between a traditionally patched program and a faux patched program. This suggests that it is more resource intensive to analyze a faux patched program compared to a traditionally patched program, thus analyzing ghost patches would also require more resources.

6 Discussion

Our proof of concept implementation shows that the application of deception, in the form of fake patches, to software patching is feasible. Our evaluation shows that a faux patch does have an impact on exploit generation, increasing the number of branches in a program, by increasing the resources necessary

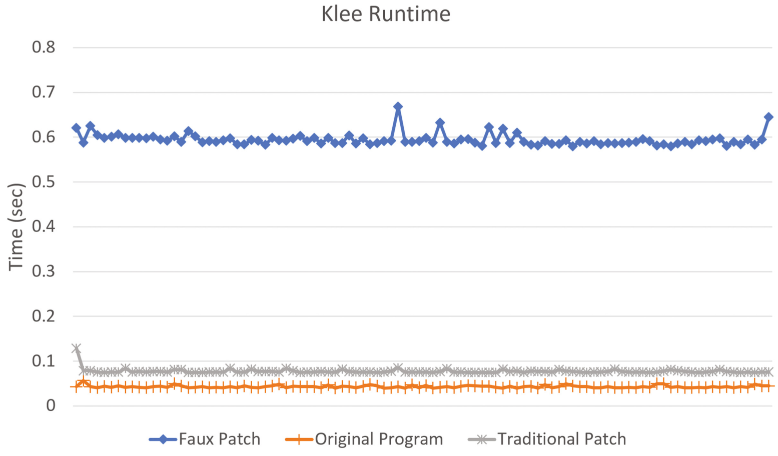

Fig. 3. Klee runtime analysis

to analyze a program. These same patches also impact a program's runtime, but this effect is not statistically significant. This suggests that deception can be used to make exploit generation using patches more resource intensive, enhancing the security of software patches. We believe that with additional research and testing, this approach, either as a standalone technique or in conjunction with other deceptive and detection methods, could impose an exponential increase in program analysis, making exploit generation based on patches an expensive operation, while only adding a minimal increase in program runtime. Our proof of concept implemented and analyzed above supports this claim.

Patch Obfuscation. There are limitations associated with ghost patches that could provide attackers an advantage in identifying fake patches or minimizing their impact on program analysis. Attackers could use exploit generation tools that perform analysis in parallel [6] to distribute the analysis load across multiple machines and optimize exploit generation. One solution is to develop fake patches that increase the length of each path in a program such that tools are unable to develop an exploit. Another solution is to implement polymorphic patches. Ghost patches can utilize randomization to create polymorphic patches that can be distributed based on different heuristics (i.e. based on region, OS version, or staggered by time). The non-deterministic nature of a polymorphic ghost patch could make exploit development more difficult because the same patch would not be applied to each end system. In this case, the traditional patch would also have to be altered for each patch instance to prevent attackers who utilize multiple instances of a patch to expose the legitimate vulnerability.

Based on our observations, traditional patches for input validation vulnerabilities detect malicious input and *return gracefully* from the function. This prevents a compromise, but when viewing a binary diff, searching for differences that add *return* commands could be an identification technique. Applying obfuscation to fake and legitimate patches or to the function being patched could increase the difficulty in distinguishing between each type of patch. Future work will explore obfuscation techniques to make code more difficult to understand [13] and control flow more difficult to evaluate [17].

Active Response Patches. Based on the *non-interfering* property, faux patches should not alter the semantics of the program; the verify step will expose that fake patches do not alter program behavior. Thus, at worst, a brute force approach could expose the vulnerability by analyzing program behavior for each path in a program and identifying which change a program's behavior.

One solution is to use the active response technique for legitimate patches. Active response patches prevent a vulnerability from being exploited but respond to exploits using the same response as an unpatched program. The response could return sanitized data from the actual machine or transfer execution to a honeypot environment [2]. This masking would increase the resources necessary for dynamic analysis tools to identify unpatched systems. Further research will develop techniques that hinder or prevent exploit verification.

Approach Limitation. Another limitation is that based on our experiments, ghost patches only have a dynamic analysis impact when there are multiple *store* operations within a program's intermediate representation (i.e. operations that includes an = sign). Programs that use standard functions (i.e. *memmov*, *memcpy*) to assign values semantically perform the same operation, but are represented differently syntactically, and thus a fake patch cannot be applied.

Adding new lines of code also could add unexpected vulnerabilities. The faux patch code is like any other code that could have a vulnerability. Ghost patched code could also be attacked. Providing attackers with additional paths that could be attacked could result in a denial of service type of attack that slows overall program runtime which could impact the machine's performance.

Future work will extend our tool to compile and add fake patches to more complex code. Additional testing will give insight into the effectiveness of ghost patches. We believe because of the simplistic nature of our approach (i.e. adding conditional statements using the store instruction), its' statistically significant increase in program analysis time will not be lost. We also believe that because of the fake patch if-statement body's code, the difference in runtime will not be statistically significant.

7 Conclusion

This work proposed, implemented and evaluated ghost patching as a technique to mislead attackers using patches to develop exploits against input validation vulnerabilities. We discuss fake patch properties as well as analyze a proof of concept using LLVM. Through experimentation, we found that fake patches add latency to program runtime that is not statistically significant while adding a statistically significant amount of latency to program analysis. If used by program developers as they develop patches for security flaws, we believe faux patches could disrupt the exploit generation process, providing more time for end users to update their systems.

Acknowledgements. The authors would like to thank the anonymous reviewers for their comments and suggestions and specially thank Breanne N. Wright, Christopher N. Gutierrez, Oyindamola D. Oluwatimi and Scott A. Carr for their time, discussion and ideas. The National Science Foundation supported this research under award number 1548114. All ideas presented and conclusions or recommendations provided in this document are solely those of the authors.

References

1. Almeshekah, M.H., Spafford, E.H.: Planning and integrating deception into computer security defenses. In: Proceedings of the 2014 Workshop on New Security Paradigms Workshop, pp. 127–138. ACM (2014)
2. Araujo, F., Hamlen, K.W., Biedermann, S., Katzenbeisser, S.: From patches to honey-patches: lightweight attacker misdirection, deception, and disinformation. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 942–953. ACM (2014)
3. Araujo, F., Shapouri, M., Pandey, S., Hamlen, K.: Experiences with honey-patching in active cyber security education. In: 8th Workshop on Cyber Security Experimentation and Test (CSET 2015) (2015)
4. Bashar, M.A., Krishnan, G., Kuhn, M.G., Spafford, E.H., Wägstäff, Jr., S.S.: Low-threat security patches and tools. In: International Conference on Software Maintenance, pp. 306–313. IEEE (1997)
5. Bowen, B.M., Hershkop, S., Keromytis, A.D., Stolfo, S.J.: Baiting inside attackers using decoy documents. In: Chen, Y., Dimitriou, T.D., Zhou, J. (eds.) SecureComm 2009. LNICSSITE, vol. 19, pp. 51–70. Springer, Heidelberg (2009). doi:10.1007/978-3-642-05284-2_4
6. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: techniques and implications. In: IEEE Symposium on Security and Privacy, pp. 143–157, May 2008
7. Cadar, C., Dunbar, D., Engler, D.R., et al.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI **8**, 209–224 (2008)
8. Collberg, C., Nagra, J.: Surreptitious Software. Addision-Wesley Professional, Upper Saddle River (2010)
9. Collberg, C.S., Thomborson, C.: Watermarking, tamper-proofing, and obfuscation-tools for software protection. IEEE Trans. Softw. Eng. **28**(8), 735–746 (2002)
10. Coppens, B., De Sutter, B., De Bosschere, K.: Protecting your software updates. IEEE Secur. Priv. **11**(2), 47–54 (2013)
11. Crane, S., Larsen, P., Brunthaler, S., Franz, M.: Booby trapping software. In: Proceedings of the 2013 Workshop on New Security Paradigms Workshop, pp. 95–106. ACM (2013)
12. Dewdey, A.K.: Computer recreations, a core war bestiary of virus, worms and other threats to computer memories. Sci. Am. **252**, 14–23 (1985)
13. Dolan, S.: mov is turing-complete. Technical report 2013 (cit. on p. 153) (2013)

14. Kanzaki, Y., Monden, A., Collberg, C.: Code artificiality: a metric for the code stealth based on an n-gram model. In: Proceedings of the 1st International Workshop on Software Protection, pp. 31–37. IEEE Press (2015)
15. Lattner, C.: The LLVM Compiler Infrastructure. University of Illinois, Urbana-Campaign (2017)
16. Mitnick, K.D., Simon, W.L.: The Art of Deception: Controlling the Human Element of Security. Wiley, New York (2011)
17. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Twenty-Third Annual Computer Security Applications Conference, 2007. ACSAC 2007, pp. 421–430. IEEE (2007)
18. Mosher's Law of Engineering: Top 50 Programming Quotes of All Time. TechSource (2010)
19. Oh, J.: Fight against 1-day exploits: diffing binaries vs anti-diffing binaries. Black Hat (2009)
20. Ben Salem, M., Stolfo, S.J.: Decoy document deployment for effective masquerade attack detection. In: Holz, T., Bos, H. (eds.) DIMVA 2011. LNCS, vol. 6739, pp. 35–54. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22424-9_3
21. Spafford, E.: More than passive defense. CERIAS (2011)
22. Stoll, C.: The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage. Simon and Schuster, New York (2005)
23. Symantec: Internet security threat report. Technical report, Symantec (2016)
24. Udupa, S.K., Debray, S.K., Madou, M.: Deobfuscation: reverse engineering obfuscated code. In: 12th Working Conference on Reverse Engineering, p. 10. IEEE (2005)
25. Wang, C., Suo, S.: The practical defending of malicious reverse engineering. University of Gothenburg (2015)
26. Whaley, B.: Toward a general theory of deception. J. Strateg. Stud. **5**(1), 178–192 (1982)
27. Yokoyama, A., et al.: SandPrint: fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (eds.) RAID 2016. LNCS, vol. 9854, pp. 165–187. Springer, Cham (2016). doi:10.1007/978-3-319-45719-2_8
28. Yuill, J.J.: Defensive computer-security deception operations: processes, principles and techniques. ProQuest (2006)
29. Yuill, J., Zappe, M., Denning, D., Feer, F.: Honeyfiles: deceptive files for intrusion detection. In: Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, pp. 116–122. IEEE (2004)