Booby Trapping Software

Stephen Crane
sjcrane@uci.edu

Per Larsen
perl@uci.edu

Stefan Brunthaler
s.brunthaler@uci.edu

Michael Franz
franz@uci.edu

Donald Bren School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

ABSTRACT

Cyber warfare is asymmetric in the current paradigm, with attackers having the high ground over defenders. This asymmetry stems from the situation that attackers have the initiative, while defenders concentrate on passive fortifications. Defenders are constantly patching the newest hole in their defenses and creating taller and thicker walls, without placing guards on those walls to watch for the enemy and react to attacks. Current passive cyber security defenses such as intrusion detection, anti-virus, and hardened software are not sufficient to repel attackers. In fact, in conventional warfare this passivity would be entirely nonsensical, given the available active strategies, such as counterattacks and deception.

Based on this observation, we have identified the technique of booby trapping software. This extends the arsenal of weaponry available to defenders with an active technique for directly reacting to attacks. Ultimately, we believe this approach will restore some of the much sought after equilibrium between attackers and defenders in the digital domain.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

Keywords

booby traps; active defense; intrusion detection; compilers

1. MOTIVATION

The status quo in cyber security puts defenders at a disadvantage over attackers: the attacker only has to find one way in, while the defender needs to guard multiple points of exposure, some of which may be unknown. Even a defender with vastly superior resources is still at risk from unknown bugs [3]. The current context in which we use software greatly amplifies this asymmetry. Two key factors multiply the attacker's advantage:

1. *Software Monoculture*: Hundreds of millions of computers run identical versions of popular software, such as Microsoft Windows, Acrobat Reader, Adobe Flash, and modern web browsers. This monoculture favors attackers disproportionately, since the exact same attack vector will be effective on large numbers of targets. Moreover, since the attacker can easily replicate the environment which he or she will eventually execute the attack against, the attacker can develop, debug, and test the attack before releasing it into the wild.

2. *Passive Defense Strategies and Tactics*: Aside from network-based defenses, the defenses deployed on the computers that are the actual targets of attacks are mainly *passive*. We define *passive* defenses as anything which makes an attack harder to accomplish but does not react automatically during an attack. This includes traditional defenses such as firewalls, hardened secure software, and scans for resident malware after infection.

Biologically-inspired defenses leading to software diversity have recently started to address the first of these factors. This is still work in progress, with solutions ranging from simple approaches such as memory layout randomization to more ambitious ones that actually diversify the binary code of application programs. Unfortunately, some approaches to binary code diversity tend to restrict possible input programs. For example, current code diversification techniques do not extend to programs that involve self-modifying code, such as just-in-time compilers. This means that there is further room for innovation in this area.

The second factor favoring the attacker, the passive nature of current defense mechanisms on target computers, has received much less attention so far. In contrast to *passive* defenses, we define *active* defenses as countermeasures triggered as a direct reaction to an attack. Intrusion prevention systems (IPS) which together with a firewall actively block network connections when the IPS detects an ongoing attack is a good example of such *active* defenses. Part of the reason for the current lack of active defenses is that an active solution will consume some resources. Frequently, users are unwilling to deploy heavyweight schemes. Additionally, active defenses may have high false positive rates and detrimental effects for

NSPW'13, September 9–12, 2013, Banff, AB, Canada.
Copyright 2013 ACM 978-1-4503-2582-0/13/09 ...$15.00.
http://dx.doi.org/10.1145/2535813.2535824.

legitimate users, making system administrators even more reluctant to deploy them.

This paper describes a new defense paradigm that address both factors. Rather than just trying to build stronger defenses, we suggest also introducing active responses to discourage the attacker. Our technique protects software without human intervention using either compile-time or load-time rewriting to add active, diversity-based defenses with no run-time overhead. In particular, we focus on extending the compiler or loader to weave dormant, active defenses into generated binaries. Among other capabilities, these active defenses can discourage brute-force attacks by costing the attacker time or other resources Since the defenses that our system weaves into the target code are not executed during normal operation, the protected programs run at full speed. We use the term "arming" a program to refer to this process of automatically adding these active defenses to strengthen a program.

We have identified one such active defense which we call the "booby trap." We define booby traps as code providing active defense that is only triggered by an attack. These booby traps do not implement program functionality and do not influence its operation—in fact, the program does not know about its own booby traps and under normal operation cannot trigger them. We propose to automatically insert booby traps into the original program code during compilation or program loading. Whenever an attack triggers one of the booby traps within the program, the trap instantly knows that the software is under attack, and is in a position to adequately react to the threat (see Figure 1). For example, a booby trap might perform advanced forensics to identify an attack in real-time to facilitate a deceptive response. In Section 4 we describe further research directions and new opportunities to engage attackers.

The use and effect of cybersecurity booby traps bears striking resemblance to their real-world equivalents. First, a booby trap signals the position of the enemy to the defender. Second, it is in the interest of the defender to place booby traps in locations with a high likelihood of being triggered by the enemy. Third, booby traps affect the mindset of the enemy: they demoralize attackers, keep them stressed, make them cautious instead of aggressive, and slow down the enemy's movement. We anticipate that all of these effects hold true in the digital domain, too.

Finally, we think that booby trapping is a transformative approach to cybersecurity, since it provides new options to defenders. By arming a program, we fundamentally alter the terrain of cyber warfare, since active, direct response to intrusion gives defenders an equalizing advantage. Defenders need not constrain themselves to preparing for attacks or cleaning up after them, instead they can automatically and reliably respond during an attack. These responses are not triggered by heuristics and observation, as the few current active approaches are, but rather as a direct side-effect of the attack. This means that defenders can pursue more aggressive reactions, with the assurance that there really was an attempted attack. This ability to respond tips the scales of cyber war back towards the defender.

Summing up, our contributions are:

- We propose a new active cyber defense tool, *booby traps*, which will help to correct the current asymmetry in cyber warfare initiative. In Section 3 we discuss a concrete example of how defenders could insert booby traps in the context of code-reuse attacks.

- We provide examples that demonstrate how cyber booby traps can actively defend software by enabling responses such as providing additional information to defenders, preventing and recovering from attacks, and disincentivizing attackers. In Section 4 we enumerate and discuss useful responses particularly suited to booby traps.

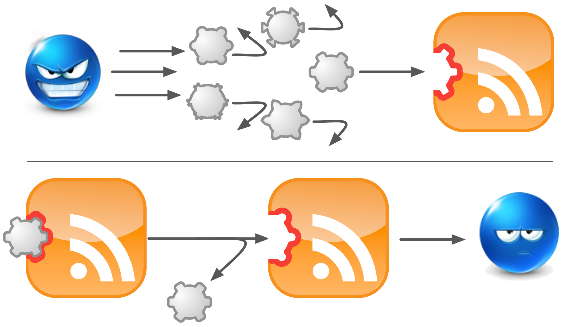

Figure 1: Software diversity forces the attacker to bombard targets to find a working exploit. Even if the attacker manages to find an exploit that "sticks," active defenses now trigger a reaction in the target.

2. BACKGROUND

While the concept of booby traps is an applicable defense tactic for many types of attacks, in this paper we will focus primarily on applying booby traps to protecting against code-reuse attacks, which are an increasingly important threat. Throughout this paper we will also assume a remote attacker model, where the attacker does not have direct access to the process under attack. Thus, attacks from the same machine such as software or hardware tampering are out of scope for our defenses, which assume that we can hide booby traps from the attacker. To give context to this discussion, we must first explore code-reuse attacks and relevant defenses.

To prevent classic code injection attacks, modern operating systems use security mechanisms, such as $W \oplus X$ and code signing. This led to the development of attacks which circumvent these protection mechanisms entirely by reusing existing code. Code-reuse attacks include "return-to-libc" [35] attacks which reuse functionality present in standard libraries and "return-oriented programming" (ROP) [40], which reuse short code sequences, *gadgets*, ending in a return instruction or instructions with a similar effect. ROP techniques are very powerful since virtually every targeted binary contains code with the prerequisite gadgets. Not only is ROP supported by attack construction toolkits, but it was also implicated in the Stuxnet attacks [33] and used to compromise a hardened electronic voting machine [13].

Attacks using this technique "thread through" code snippets contained in the gadgets by using the program stack in an unconventional manner. The attacker carefully arranges the addresses of the gadgets to be executed in sequence onto the stack. Unlike regular subroutines, characterized by `call`-`return` pairs, gadget sequencing in a return-oriented attack

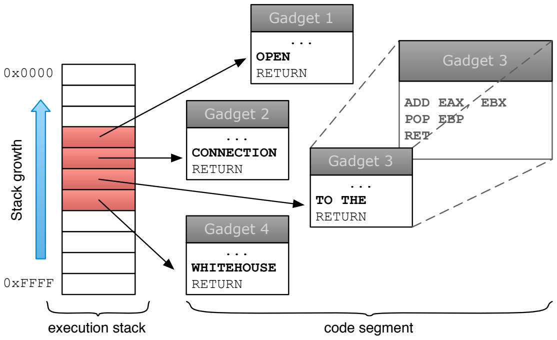

Figure 2: Attack using return-oriented programming. The attacker arranges gadget addresses on the execution stack. Code injection is avoided by reusing existing code already loaded in the process.

uses only the return mechanism. The attacker begins by diverting execution to the first gadget, updating the stack pointer in the process. The return instruction at the end of the gadget transfers the control flow to the next gadget, and so on. Consequently, the stack pointer takes the role of the instruction pointer in ROP attacks. Figure 2 illustrates a return-oriented attack.

2.1 Defending against Code-Reuse Attacks

Preventing ROP attacks is difficult because the attacker can find sufficient code to reuse in almost any non-trivial binary. Attackers generally also have access to the exact binary that is running on the target, allowing them to devise a reliable remote attack.

Fortunately, there are several ways to prevent return-oriented attacks from succeeding. First, note that the control-flow must be hijacked by an attacker to execute the gadgets. Second, the control flow resulting from an attack is unusual; jumps into the middle of functions and returns occur with much higher frequency relative to "legitimate" control-flows. Third, successful ROP attacks require gadgets to be located at predictable addresses known to the attacker. As a result, we can disrupt ROP attacks in one of the following ways:

1. preventing control flow hijacking,
2. detecting frequent returns,
3. removing the usable gadgets; or
4. diversifying gadget locations.

Each approach is distinct in terms of the security provided, generality with respect to eligible input programs, performance overhead, and so forth. We seek an approach that preserves performance and remains fully compatible with existing code. However these goals make effectively defending against return-oriented programs challenging.

The first approach, enforcing control-flow integrity [2], is conceptually simple, but difficult to realize for many programs. Distinguishing intended from unintended control-flows may be possible for a large class of programs. This class, however, does not include important programs, such as interpreters and just-in-time compilers, which cannot be left unprotected.

Why not just remove the gadgets from the binaries then? While return-less kernels [31] have attempted to do this, this approach has been shown to be ineffective since return-oriented programming can be generalized to approaches that do not use explicit return instructions [12]. A later refinement of code-reuse attacks, jump-oriented programming, does away with the use of the stack entirely, relying on indirect jumps instead of returns [8]. While we can potentially broaden the definition of a gadget and extend current work that removes gadgets, this solution relies on details of the attack and therefore requires additional effort to extend to each new code-reuse attack.

The third approach detects the high frequency of returns resulting from executing a ROP attack [14]. While this approach may prevent basic ROP attacks, code-reuse attacks can avoid return instructions by using indirect jumps, which are also frequently executed by legitimate programs, such as interpreters.

Compatibility concerns aside, the first and third approaches share another drawback: they add computational overheads to the programs they protect. While this may not be too much of a concern in high-security deployments, the resulting increase in power consumption is highly undesirable for programs running on battery-powered mobile devices, or in data centers which are increasingly power-constrained.

2.2 Diversifying Gadget Locations

Accepting that all binaries above a certain size contain a Turing-complete set of code that attackers can reuse [26], we must diversify gadget locations. By giving each end-user an unique binary, attackers no longer know where to find the gadgets in target binaries. This software diversification approach has several desirable properties. The compilation process can randomize gadget locations which avoids adding computational overhead to the protected programs. Consequently, this approach is suitable for all programs irrespective of whether they are running on an embedded device, a personal computer, or on a server in a data center. Even more important than performance, we think such diversification is compatible with virtually all input programs, including interpreters and just-in-time (JIT) compilers. This property follows from the fact that programs — apart from malware — do not rely on where code is located in a binary; the code and data layout happens at the discretion of the compiler, which exploit this to optimize for time or space. We see this degree of freedom as an opportunity to effectively enhance security through artificial software diversity.

Previous research in diversity has shown that fine-grained code diversification using NOP insertion incurs very low overhead (approx. 1%) [25]. By diversifying the code, we can be confident that a remote attacker will not be able to accurately predict the locations of gadgets he or she needs to use. If the attacker does not expect to attack a diversified code image, an attack against the expected single "canonical" binary will certainly crash due to invalid instructions or incorrect behavior. We expect this situation if only some high-security targets choose to specially compile their software with diversity. However, if all users receive diversified binaries, we still expect that crafting a code-reuse attack is not possible without exfiltrating the particular binary under attack.

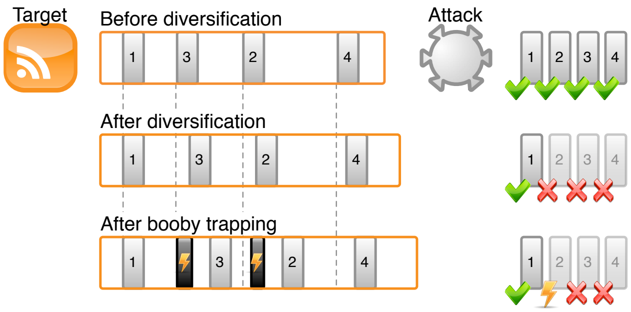

Figure 3: After diversification, the attack is disarmed since gadgets have moved from their expected locations. After booby traps are weaved into the binary, the attack is detected and the system reacts to the anomaly.

3. CYBER BOOBY TRAPS

As discussed previously, a booby trap provides an active defense triggered directly by an attack. Booby traps could be useful to defend at many different levels of the software stack. As an example in the web domain, booby traps could be inserted in web frameworks to protect against attacks which attempt to use functions or SQL statements which are not normally used by the framework. We could also insert booby traps into the operating system to trap syscalls or OS interaction which we guarantee the system will never use during normal operation.

However, we will focus on low-level booby traps as a running example in this paper. By extending the idea of artificial software diversity, we can create booby traps for code-reuse attacks. Attack code which inadvertently lands in a booby trap will trigger active defenses, which we describe in Section 4. The following sub-sections detail research questions we need to address before cyber booby traps enter the arsenal of defenders. Specifically, we describe what code-reuse booby traps are, and then determine when and where booby traps should be inserted for the greatest probability of catching an attack.

3.1 What to add?

By diversifying code as described above, we are confident that a realistic ROP attack will crash at some point when it tries to call a gadget which is broken or no longer in the expected position. However, by inserting booby trap hooks into the binary while diversifying, we can catch the attack and react to it directly. When the attack redirects execution to what the attacker thought a useful gadget, but is now a booby trap, the booby trap takes over execution and responds to the attack. By inserting booby traps where the attacker expects a gadget, or simply sprinkle traps throughout the binary so that any attack has some probability of triggering a trap instead of executing a gadget, we can reliably catch attacks while they occur. These booby traps will then pass execution to a handler which can actively respond to an attack in real-time and provide additional security to disincentivize the attacker from blindly attempting additional code-reuse attacks (see Section 4 for discussion of possible responses).

Additionally, we must decide what these handlers actually do when triggered. What makes a suitable booby trap, and who supplies the booby trap code? Obviously, an attacker should not be able to use a booby trap to mount an attack, which means that we have to either find a restricted set of eligible instructions to be used for booby traps, or supply hardened booby trap code ourselves. Either way, we have to make sure that it does not compromise security. An interesting direction to explore here is to enable programmers to supply domain specific booby trap code, which is then automatically hardened and weaved into the application code.

3.2 When to add?

After we determine how the booby traps function, we must decide when and how to best insert them. Essentially, we think that there are two options to arm a program:

1. inserting code at compile-time,

2. inserting code at load-time, or

As is usually the case, there are several trade-offs involved in deciding which technique to use. Since each has its advantages and disadvantages, we shall now explore both techniques to offer a complete and applicable solution.

Compile-time Insertion.

Arming at compile-time has the benefit that it happens ahead of time, i.e., we have greater flexibility to diversify and choose suitable booby traps to insert into a program. The compiler has full control of all code emitted and can place booby trap code at arbitrary locations. This process is far simpler and more practical than trying to rewrite existing binary code, which may not even be automatically disassemblable, but requires the availability of source code.

Compiler diversity can create a nearly infinite number of versions of software, so we can give each end-user a custom diversified and armed binary. However, this affects the release and distribution of programs, as publishers cannot realistically ship software on a physical medium anymore. Fortunately with the growing popularity of digital software distribution, we do not need to worry about this difficulty. A previous work by Franz [22] describes this paradigm shift in more detail.

Load-time Insertion.

If we diversify and arm a program at load-time, we could still ship a single binary to the users. Inserting code at load-time could provide even higher security guarantees, as not even an informed insider would be able determine what the program layout is without access after it is loaded. Further, it would extend our techniques to programs whose source code is not readily available. In this case, however, we would also have to properly handle all indirect-branches and make sure that they get properly "redirected" to the intended branch target. One way to optimize this detour away would be to add a table listing all indirect branch targets when compiling the program. Research in the area of load-time diversity by Wartell et al [46] may be a starting point which booby trap arming at load-time could be based upon.

So, we see that there is no clear best approach, but rather varying points on the same spectrum. Furthermore, diversifying/arming at both times, i.e., at compile-time and load-time, results in a system offering higher security in face of potentially compromised parties: compile-time arming protects against a compromised or disabled load-time diversifier, and load-time arming protects against a compromised code producer or use of an insecure transmission medium.

3.3 Where to add?

If we choose to insert booby traps either at compile-time or load-time, we need to address the question of *where* we actually insert our trap code. Guiding this decision is the goal of inserting in locations an attacker will execute, but where normal execution will never trigger the trap.

We expect that finer grained insertion will result in stronger security, although at the cost of slightly slower performance. For example, inserting a NOP slide ending in a booby trap within an existing basic-block is going to increase its size, decreasing instruction cache utilization and thus increase instruction cache misses. Whenever we insert a booby trap into a basic block, we need to make sure that the program's control flow does not reach it. This can be done by breaking the basic block in sub-blocks and jumping past the booby trap. Using this technique, we expect to reduce performance penalties. We expect that this fine-grained level of insertion does positively affect security, as we can insert booby traps in all places that are attractive to attackers.

To insert booby traps in positions which an attacker will likely execute, we can find all possible ROP gadget locations and insert traps at these locations, displacing the existing gadgets. However, this assumes a known attack model, which is generally not the case. To properly combat new and evolving threats, we recommend inserting traps in random locations throughout the binary so that any code-reuse attack will have some probability of triggering a trap.

4. ACTIVE RESPONSES

Since attacks directly trigger booby traps, which then run inside the program space under attack, booby trap handlers are in a unique position to react to attacks. A booby trap is the fastest possible reaction to an attack, since it runs during the attack itself. Booby traps also have full access to the program and attempted exploit, which allows them to modify the execution of both the program and attack. Finally, booby traps are an integral part of the application code itself, and as such cannot be easily disabled by an attacker, especially if the operating system verifies that binaries are properly signed. Based on this unique position, we think that there are many interesting ways to use booby trapping to increase the cost of attacks.

When considering responses, we must first determine how much information we will allow to the attacker about our response. Stealthy responses allow defenders to maintain an advantage by responding as the attacker expects and therefore not leaking knowledge that the attack occurred. We can engineer many responses to either be stealthy or not, although a few that we suggest below will by nature betray their existence.

Recovery

Since we have access to the running program in the booby trap, we can attempt to recover from the attack, rather than simply crash. For certain high-availability services, it is more desirable that the program stay running, even if the system cannot guarantee program correctness. A recovery handler could use standard software fault tolerance techniques such as checkpointing, or more domain specific techniques such as inferring what data was corrupted in the attack and synthesizing replacement data. Additionally, we can use the booby trap to re-activate programs, memory protection flags, and other security mechanisms that the attack disabled by reverse engineering the attack payload.

Recovery will generally leak information back to the attacker, although we are unsure how attackers might interpret this information. Depending on when the attack triggers the trap, we may recover before the attacker actually knows that the attack has partially succeeded. However, this recovery must not alter correct execution of the software, or the attacker can observe unexpected side-effects.

Version Flux

Take a snapshot of the current program state and re-start another program instance with this state. This is particularly effective if the new program is a program containing a different set of diversifications and/or booby traps. There are several compelling applications of mixing program diversification with execution, for example, we could supply a custom program loader which chooses one program out of many diversified/armed ones at random. In such a situation, even insider knowledge might not be sufficient to sabotage the computing infrastructure, because the attacker cannot deduce which of the variations is currently running, or will be running on the next launch.

Version flux naturally leaks information to the attacker, since a new version of the software ideally behaves differently in response to the same attack. However, we can introduce noise into this leak by switching software versions randomly without detecting an attack.

Honeypots

Honeypots are network machines specifically configured to be vulnerable to attack in order to gather information about new exploits and tactics. Booby traps are ideal for this purpose, since they can provide detailed information about attacks, especially the initial exploitation vector. By using advanced forensics capabilities (see Section 4) armed binaries can report exactly what the attack is doing in real-time without requiring further execution in a virtual machine. We believe that armed binaries will be a very useful new tool to existing honeypot research such as the Honeynet project [43].

Honeypots are ideal for implementing stealthy booby traps. We expect that hiding most responses from the attacker will require the use of some sort of a honeypot to simulate the attack completing successfully.

Feedback Directed Instrumentation

Until now, we have only talked about the technicalities of inserting active defenses, i.e., how, where, and what. Doing this, however, at all places that could be a potential attack is a conservative approach. If we assume a software monoculture, as we have today, where all binaries of a single version of an application are identical, we can insert booby traps where known gadgets would be located in the "expected" binary. While this increases security if applied for only a minority of users, attackers will not have an "expected" binary if these techniques are widely utilized. In this situation, we can insert booby traps of various sizes randomly into the code, and probabilistically catch an attack targeting some particular binary (assuming we protect the binary itself from exfiltration or diversify at load-time).

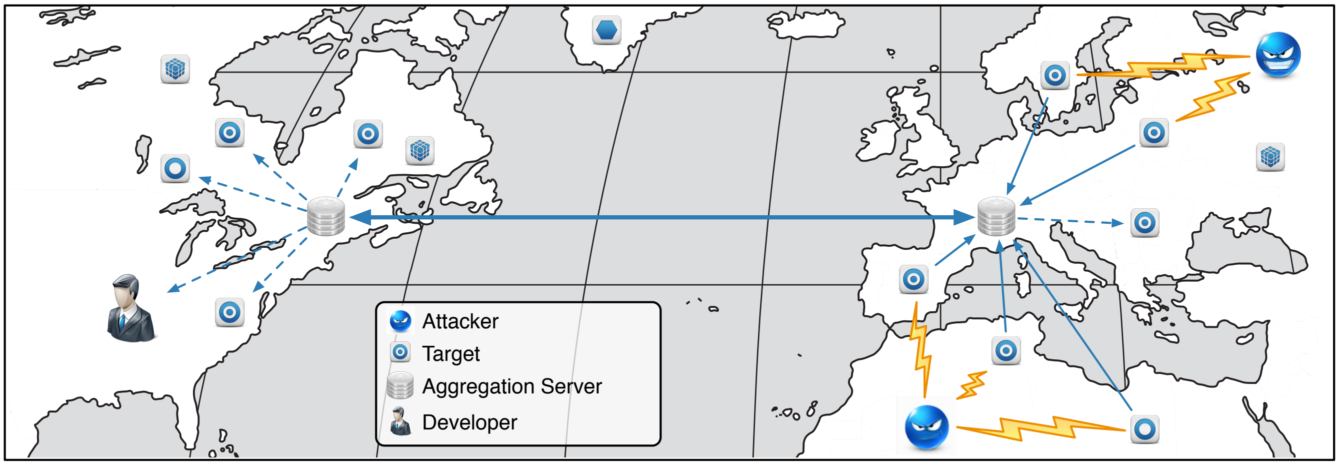

Figure 4: When attacked, armed binaries report the attack to aggregation servers that monitor the attacks in real-time. Defenders can use the early warning capabilities to alert developers and cause potentially vulnerable systems to take preventive action ahead of the attack.

Whenever we discover classes of new attacks, we can add a dedicated analysis phase and make sure that targeted locations in the "expected" binary are prepared to trigger booby traps in our version. Arming programs in this way allows us to make better guesses as to where to actually add booby traps and where a simple `NOP` pad suffices. It also allows us to guarantee that no gadgets survive (remain usable) between any two programs in a population of diversified binaries. Finally, by also using feedback obtained from a performance profiler, we can also avoid instrumenting the most frequently executed code as demonstrated by Homescu et al. [25].

Enhanced Forensics

Complimentary to improved accuracy, which is possible by using adaptive techniques as the previous paragraph describes, we can use booby traps to trigger a host of forensic analyses. The purpose of such analysis is to determine the origin and signature of the attack. Since it is difficult to determine if a program crash is caused by an attack or simply a coding error, triggering forensics in the early stages of an attack improves our opportunities to distinguish the two cases.

Since attacks trigger booby traps while they are still running we can deeply inspect the memory of a running attack. For example, return-oriented programming uses the return-stack of the native machine to manage the control of the attack, similar to an instruction pointer. When the ROP-attack triggers a booby trap, we can walk the stack of the program in reverse order and use a *gadget look-up table* to find which words hold data, and which (most likely) point to gadgets. Whenever we have looked at enough data points, it is likely that we have "recovered" the original stack that the attacker prepared. At this point, we can actually run analysis on the attack, and use well-known techniques, such as signatures, to identify how the actual attack works.

While some of this is possible with conventional forensics techniques, triggering these techniques from a booby trap lets us catch an attack in progress, and thus allows us to inspect the initial attack vector. Using booby traps we can do this without costly deep packet inspection and storage to replay attacks after-the-fact. Since booby traps trigger in real-time, we get immediate feedback on techniques and intentions of the initial attack vector. We can use these features to our advantage in the following way: First, we could use forensics analysis results, possibly based on the signatures to find out the entry point and possible goals of the attack. Next, we can potentially deceive the attacker with incorrect knowledge in real-time, or move the whole attack to a safe environment, e.g., to a virtual machine, while it is still running. Finally, we can drive or validate the feedback directed insertion process using forensics information to prevent future similar attacks.

We think that deception based on advanced forensics is a promising application for booby traps, since this allows defenders to respond stealthily to the attack. Cohen has already done significant work in the field of deception, and has shown that deception is an extremely powerful technique [16]. Using booby traps to trigger deceptive campaigns can especially provide low-level deception that the attack succeeded when in fact it did not. Real-time forensics provides the details of the attack, allowing defenders to possibly simulate the attack in a secure environment. Defenders then have full control over the attack and can construct a simulated environment to deceive the attacker.

Counterattack

Although counterattacking (colloquially termed "hacking back") is rife with dangers, pitfalls, and questions of legality, we believe that booby traps might be a good point to launch counterattacks from. Previous literature has discussed the legality and morality of counterattacks [29, 20], and we defer this discussion to more qualified experts on the subject. Instead we simply address the role that booby traps might play in facilitating counterattacks.

Booby traps could serve as an automatic start to the counterattack process. As an automated response, booby traps could quickly engage the attacker before he or she even realizes that the attack has failed. This might allow the counterattack to catch the attacker unaware and before he or she has a chance to retreat and abandon the source of the attack. However, since counterattacking is risky, any automated response would need to use the forensics techniques we touched upon earlier to verify the trigger was a real attack rather than a software bug. In addition, counterattack, or even counter-intelligence, betrays that the booby trap detected the attack, which is a large disadvantage. We imagine that booby traps might serve as only a starting point in a counterattack campaign, perhaps providing initial port scan or automated vulnerability scan results to a human operator who would then continue the campaign. However, with automated attack toolkits such as Metasploit [1], it is certainly possible that an automated counterattack may gain access to the offending system. While counterattacking is still under debate, there may be contexts where it is a valid tactic and booby traps could provide the perfect starting point to trigger such counterattacks.

During the workshop we also discussed the possibility of internal cooperation for counter-intelligence. If both the victim and attacking machines are in the same administrative domain, booby traps could trigger a mechanism installed on all internal machines, giving direct access to the attacking machine for forensics. This would allow the booby trap at the very least to instantly follow the attack trail until it left the organization.

Cooperative Situational Awareness

In the battlefield, access to relevant information regarding the situational awareness plays a crucial role. Consequently, military leaders depend on networked and interconnected systems to use these information to their tactical advantage. We think that duplicating this approach in the digital domain results in a similar competitive edge. Binaries could be armed so that triggering a booby trap causes the target binary to broadcast information about the ongoing attack to other interested parties. For instance, cloud computing resources could aggregate attack reports and thus let defenders track and locate attackers in real time (see Figure 4.) Probabilistic and machine learning techniques let us identify causal events and detect denial of service and decoy attacks in such a cooperative situational awareness system.

We think that defenders will benefit in multiple ways. First, this system can further automate the task of tracking and locking down malicious IP traffic and identifying the ISPs hosting the attacker-controlled servers. We can notify local law enforcement much earlier relative to current practice and simultaneously collect more evidence, too. Second, the triggering of booby traps in a single armed binary could provide an early warning to all other armed binaries created from the same input program. This means, for example that we can start *preventive re-diversification* of the binary started even *before* the attack reaches the hosting computer. Finally, the alerted hosts can take steps to confuse the attackers by sending them decoy information to increase the likelihood of their apprehension.

5. RELATED WORK

There is a multitude of related work with respect to software diversity and defeating return-oriented programming. Additionally, there has been some research into active responses to intrusion detection. However, to the best of our knowledge there has not been any research on active code-based cyber-defenses, such as our booby traps.

Code-Reuse Attacks

Return-oriented programming was initially demonstrated by Shacham [40] in 2007 for the x86 architecture. ROP is a generalization of the return-to-libc attacks first described by Nergal in 2001 [35] and an evolution of the "borrowed code chunks" technique described by Krahmer in 2005. Buchanan et al. [10] extended this in 2008 to a generalized version of ROP, targeting fixed-width instruction set architectures.

Several automated tools [27, 39, 26] scan a given binary, produce a set of useful gadgets, and optionally a payload that uses those gadgets. These tools build a database of gadgets that is, in most cases, Turing-complete. However these tools require that the attacker have a copy of the binary running on the target system, and if we insert booby traps in locations the attacker expects gadgets, we will catch the attack.

Address Space Layout Randomization (ASLR) [38] has been proposed and implemented to prevent code-reuse attacks. Unfortunately attackers can circumvent this protection when the program leaks pointer values or brute-forcing is practical. We envision our booby traps to be complementary to ASLR, and could augment ASLR by providing an active response when an attacker tries to circumvent the randomization.

New defense mechanisms, such as return-less kernels by Li et al. [31] and frequent return detection by Chen et al. [14], were proposed to defeat return-oriented programming. The former technique aims to disrupt return-oriented programming by removing all gadgets, while the latter is a runtime technique that has high overheads without direct hardware support. In our work, we do not intend to make return-oriented programming strictly impossible, but rather prohibitively increase attackers' software exploit development cost. It is quite possible that a diversified binary has most, if not all, gadgets from the original binary, but the randomness introduced in the locations of these gadgets along with active responses when an attack triggers a booby trap render a ROP-based attack intractable.

Checkoway et al. [12] demonstrate a return-less approach that thwarts Li et al.'s [31] and Chen et al.'s [14] defenses. Instead of using the conventional approach which depends on `RET` instructions, Checkoway et al. [12] make use of certain instructions that behave like a return instruction. Similarly, *jump-oriented programming* [8] does not use the stack in any way and does not require that gadgets end in the `RET` instruction. Our approach targets these techniques, too.

Software Diversity Defenses

Cohen's seminal paper [15] on operating system protection by leveraging program evolution anticipates much of the development in what we now call artificial software diversity. Consequently, it is safe to say that this work motivates subsequent research in automated software diversity in general. Cohen describes several program evolution techniques, for example he describes adding garbage computation to another program using a source-to-source compiler, while we insert useful traps which the program does not execute under normal circumstances. Forrest et al. [21] also advocated diversity to combat the currently insecure software monoculture.

In 2008 Jacob et al. [28] introduce the idea of a "superdiversifier," a compiler that performs superoptimization [32] for the purposes of increasing computer security. More recently, Giuffrida et al. [23] used a diversifying compilation scheme to protect operating systems from kernel level exploits. Their approach collects meta-data during compilation to allow re-randomization of kernel components while the system is running. In 2013 Homescu et al. [25] present compilation techniques to introduce diversification, using profiling feedback to reduce the performance impact of introducing additional `NOP` instructions. This work could be extended to incorporate our compilation-based approach to booby trap insertion.

In 2012 Hiser et al. [24] and Wartell et al. [46] presented their approaches to introduce artificial software diversity at load-time without the need for source code. We believe that our load-time arming techniques are compatible with both approaches. Pappas et al. [37] also describe a technique for introducing diversity at load-time, but their approach constrains diversification to techniques which do not reposition code, and therefore cannot insert additional code for booby traps.

Instruction set randomization [7, 30] (ISR) is another interesting approach towards introducing diversity. In 2005, Sovarel et al. [42] describe a technique that illustrates some of the limitations of ISR. In 2009, Williams et al. [47] present an updated ISR implementation using virtual machines and a stronger cryptographic basis by replacing the original encryption algorithm with AES.

Other Prevention Techniques

Other approaches to the prevention of return-oriented programming attacks and arbitrary-code execution attacks are based on preventing the attacker from either taking control of the program execution or making changes in disallowed sections of memory. Since most attacks begin by taking advantage of a vulnerability such as a *buffer overflow* or *printf format string vulnerability* in order to overwrite some memory address with a value chosen by the attacker, researchers have dedicated significant research work to techniques meant to prevent these attack vectors. A significant number of attacks simply overwrite the return address on the stack directly. StackGuard, by Cowan et al. [17] is a simple transformation to stack frames that can be used for security. StackGuard places a value called a *canary* on the stack in between the stack variables and the activation record that must survive an integrity check once the function terminates. Other transformations [44] protect against stack-based buffer overflows.

Another approach, called *software fault isolation*, restricts the allowed regions of memory that the program can access. These ranges are also separated into *code* and *data* regions, which have $W \oplus X$ restrictions. Isolated programs are also prevented from jumping into the middle of an instruction on CISC machines, forcing any indirect jumps to be aligned to the start of an intended instruction. The PittSFIeld [34] software fault isolation system inserts `NOP`s at the end of each basic block to enforce alignment of jump targets. A verifier checks input programs before execution and prevents them from being executed if not all memory accesses and control flow transfers are instrumented as described. The Native Client [48] project extends PittSFIeld using x86 segmentation and other techniques to improve performance.

Abadi et al. [2] more generally describe *control flow integrity*, placing restrictions on all indirect jumps and returns so that jump targets belong to a whitelisted set. The system inserts the checks as efficiently as possible using static analysis. Because these projects perform software fault isolation and enforce control flow integrity, they have side effects that are effective defenses against return-oriented programming. However, like instruction set randomization, we are not aware of any research that shows the implications control flow integrity has on return-oriented programming. Ansel et al. [4] optimized the execution of `NOP` padding in their extension to Native Client.

In 2010, Onarlioglu et al. present a set of techniques that "de-generalize the [ROP] threat to a traditional return-to-lib(c) attack." [36] Their technique allows for comprehensive protection against (jump-) and return-oriented programming attacks at the expense of adding run-time checks to the secured programs.

In 2011, Cui et al. [18] present their Symbiotes approach to secure the embedded networked devices. It is able to inject itself into Cisco routers and monitors the firmware while it is running. To create a symbiote, it is necessary to obtain a program binary and scan points where "code interception" is possible. A symbiote inserts itself at a randomly selected subset of these points and monitors the firmware operation of the router. These symbiotes are executed during normal program flow and can perform diagnostic and monitoring functionality. In contrast, our approach inserts code that is only executed when control flow diverges from normal execution due to an attack. This allows us to intercept and prevent attacks which attempt to reuse program code, while not introducing additional overhead during normal execution.

Active Defenses

Much of the closest work in defenses which we term *active* has been in the realm of intrusion prevention and response systems, which are intrusion detection systems extended with techniques to actively mitigate or repair detected intrusions. Common commercial intrusion prevention systems often include functionality to automatically block network connections detected as malicious [5]. Recently research intrusion response systems have included capabilities of attempting to trace intrusions to their sources [45]. There has also been considerable work in automatically choosing proper responses to intrusions [11, 6].

Most of the previous work in intrusion prevention and response systems is compatible with our techniques. Booby traps can directly serve as a replacement or enhancement to the intrusion detection portions of these systems. However, booby traps have the advantage of being a reliable source of attack information, since they can only be triggered by a program bug which attackers then exploit. We expect that existing IPS and IRS responses will be a good starting point to develop responses tailored to the advantages of booby traps specifically.

As a specific example of a possible active response, we highlight recovery and self-healing. There has been research into both developing patches to immunize against later attempts at the same exploit (such as [41]), as well as recovery to keep the compromised process running (such as [19]). Booby traps could directly trigger and use these responses.

Finally, there has been some work in creating active decoy documents to catch information exfiltration [9]. This is a similar to our booby traps, since this system attempts to actively track exfiltration when triggered. However, the document decoys attempt to prevent information loss through existing known channels, while we focus on protecting software against exploitation.

6. CONCLUSION

Active defenses are a critical part of a strong defense strategy, and we believe that booby traps are a great tool to provide active response. As attackers become more powerful and persistent, we, as the defenders, must find new ways to handle these attacks. Sitting idly by or monitoring for compromises while attackers bombard critical systems is no longer sufficient to fight off adversaries. However, with booby traps, defenders can respond immediately to attacks with intimate knowledge of the exploit itself and the vulnerable software.

By triggering from the attack itself, booby traps give defenders a variety of unique responses. Booby traps have the potential to allow software to recover from attacks, rather than crashing immediately. They also provide a platform for automatic and accurate forensics inside an attack. Since booby traps trigger in real-time, defenders could even automatically launch attribution or deception responses, while the attack is in progress and an open connection is still available. We are just beginning to scratch the surface of what might be possible with real-time, active responses triggered by attacks directly, but we believe that booby traps are a powerful weapon to add to the defense arsenal. We look forward to the development of new types of booby traps, especially to combat higher-level vulnerabilities, such as are prevalent in web applications.

Automatically arming software by inserting cyber booby traps will have a lasting effect on the field of cybersecurity. Booby traps are the sentries guarding existing defensive walls such as diversity. With these sentries, software armed with booby traps actively protects itself by handling attacks as they occur, allowing defenders to raise the cost and risk of attempted attacks. Using active response to raise the bar for attacks tips the balance of power in cyber war towards the side of the defender. By booby trapping software defenders can now take the initiative and fight back against incoming attacks.

Acknowledgments

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N660001-1-2-4014, by the National Science Foundation (NSF) under grant No. CCF-1117162, and by a gift from Google.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

We especially thank our reviewers, shepherd Daniela Oliveira, and all the workshop participants for their valuable contributions.

7. REFERENCES

[1] *Metasploit Penetration Testing Software.* `http://www.metasploit.com/`.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security (TISSEC)*, 13:4:1–4:40, 2009.

[3] R. Anderson. Why information security is hard - an economic perspective. In *Proceedings of the 17th Annual Computer Security Applications Conference*, ACSAC '01. ACM, 2001.

[4] J. Ansel, P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 355–366. ACM, 2011.

[5] N. B. Anuar, M. Papadaki, S. Furnell, and N. Clarke. An investigation and survey of response options for intrusion response systems (IRSs). In *Proceedings of the 10th Annual Information Security for South Africa Conference*, ISSA '10, 2010.

[6] I. Balepin, S. Maltsev, J. Rowe, and K. Levitt. Using specification-based intrusion detection for automated response. In *Proceedings of the 6th Interntional Symposium on Recent Advances in Intrusion Detection*, RAID '03, pages 136–154. Springer Berlin Heidelberg, 2003.

[7] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized Instruction Set Emulation. *ACM Transactions on Information and System Security (TISSEC)*, 8(1):3–40, 2005.

[8] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40. ACM, 2011.

[9] B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo. Baiting inside attackers using decoy documents. In Y. Chen, T. D. Dimitriou, and J. Zhou, editors, *Security and Privacy in Communication Networks*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer Berlin Heidelberg, 2009.

[10] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 27–38. ACM, 2008.

[11] C. Carver, J. M. Hill, J. R. Surdu, and U. W. Pooch. A methodology for using intelligent agents to provide automated intrusion response. In *Proceedings of the IEEE Systems, Man, and Cybernetics Information Assurance and Security Workshop, West Point, NY*, pages 110–116, 2000.

[12] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 559–72. ACM Press, 2010.

[13] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC Advantage. In *Proceedings of the 4th Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*. USENIX Association, 2009.

[14] P. Chen, X. Xing, H. Han, B. Mao, and L. Xie. Efficient Detection of the Return-oriented Programming Malicious Code. In *Proceedings of the 6th International Conference on Information Systems Security*, ICISS '10, pages 140–155. Springer Berlin Heidelberg, 2010.

[15] F. Cohen. Operating system protection through program evolution. *Computers and Security*, 12(6):565–584, Oct. 1993.

[16] F. Cohen. A note on the role of deception in information protection. *Computers & Security*, 17(6):483–506, 1998.

[17] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78. USENIX Association, 1998.

[18] A. Cui, J. Kataria, and S. J. Stolfo. From prey to hunter: transforming legacy embedded devices into exploitation sensor grids. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 393–402. ACM, 2011.

[19] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, page 78âĂŞ95. ACM, 2003.

[20] D. Dittrich and K. E. Himma. Active response to computer intrusions. In *The Handbook of Information Security*, volume III. 2005.

[21] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 67–72. IEEE Computer Society, 1997.

[22] M. Franz. E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism. In *Proceedings of the 2010 New Security Paradigms Workshop*, NSPW '10, pages 7–16. ACM, 2010.

[23] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Security Symposium*, pages 475–490, 2012.

[24] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, S&P '12, pages 571–585, 2012.

[25] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *Proceedings of the 11th International Symposium on Code Generation and Optimization*, CGO '13. ACM, 2013.

[26] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz. Microgadgets: size does matter in turing-complete return-oriented programming. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies*, WOOT '12. USENIX Association, 2012.

[27] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, pages 383–398. USENIX Association, 2009.

[28] M. Jacob, M. Jakubowski, P. Naldurg, C. Saw, and R. Venkatesan. The superdiversifier: Peephole individualization for software protection. In *Proceedings of the Third International Workshop on Security*, volume 5312 of *Lecture Notes in Computer Science*, pages 100–120. Springer Berlin Heidelberg, 2008.

[29] V. Jayaswal, W. Yurcik, and D. Doss. Internet hack back: counter attacks as self-defense or vigilantism? In *Proceedings of the 2002 International Symposium on Technology and Society*, ISTAS '02, pages 380 – 386, 2002.

[30] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 272–280. ACM Press, 2003.

[31] J. Li, Z. Wang, X. Jiang, M. C. Grace, and S. Bahram. Defeating Return-oriented Rootkits with "Return-Less" Kernels. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 195–208. ACM, 2010.

[32] H. Massalin. Superoptimizer: a look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS-II, pages 122–126. IEEE Computer Society, 1987.

[33] A. Matrosov, E. Rodionov, D. Harley, and J. Malcho. Stuxnet Under the Microscope, 2010. `http://go.eset.com/us/resources/white-papers/Stuxnet_Under_the_Microscope.pdf`. Accessed 04/09/2013.

[34] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*. USENIX Association, 2006.

[35] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, Issue 58, 2001.

[36] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 49–58. ACM, 2010.

[37] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, S&P '12, pages 601–615, 2012.

[38] PaX. *Homepage of The PaX Team*, 2009. `http://pax.grsecurity.net`.

[39] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Security Symposium*. USENIX Association, 2011.

[40] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the

x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561. ACM Press, 2007.

[41] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. *Proceedings of the general track, 2005 USENIX annual technical conference: April 10 - 15, 2005, Anaheim, CA, USA*, pages 149–161, 2005.

[42] N. Sovarel, D. Evans, and N. Paul. Where's the FEEB?: The Effectiveness of Instruction Set Randomization. In *Proceedings of the 14th USENIX Security Symposium*, pages 145–160. USENIX Association, 2005.

[43] L. Spitzner. The honeynet project: trapping the hackers. *IEEE Security Privacy*, 1(2):15–23, 2003.

[44] Vendicator. StackShield: A "stack smashing" Technique Protection Tool for Linux, 2000. `http://www.angelfire.com/sk/stackshield/`.

[45] X. Wang, D. S. Reeves, S. F. Wu, and J. Yuill. Sleepy watermark tracing: An active network-based intrusion response framework. In *Proceedings of the 16th International Information Security Conference*, pages 369–384, 2001.

[46] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168, 2012.

[47] D. W. Williams, W. Hu, J. W. Davidson, J. Hiser, J. C. Knight, and A. Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *IEEE Security & Privacy*, 7(1):26–33, 2009.

[48] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, S&P '09, pages 79–93. IEEE Computer Society, 2009.