CAREER: Towards Identifying and Eliminating Exploitable Software Bugs

Attackers only need to find a single exploitable bug in order to install malware, bots, and viruses on a vulnerable user's computer. Unfortunately, bugs are plentiful. For example, the Ubuntu Linux bug management database currently lists over 58,000 open bugs [56]. Specific widely-used programs such as Apache 2.0, Firefox 3.0, and the current Linux 2.6 Kernel have respectively 544 [53], 501 [54], and 1337 [55] open, non-trivial bugs. Closed-source and commercial projects are likely to have similar statistics. Thus, the question is not whether an attacker can find a security-critical bug, but *which bug the attacker will find and exploit first.*

The problem defenders face is determining which bugs are exploitable, how an attacker might exploit them, and how to safely distribute patches that fix exploitable bugs to end-users. These are the primary research questions this proposal explores. These questions are key to the long-term goal of the PI to minimize the attack window — the time from when an exploitable bug is introduced to when the software patch is applied on the end-system.

Intellectual Merit: This research will develop new methods for finding exploitable bugs and prioritizing them. This research will also develop methods for safely distributing patches that fix exploitable bugs, as well as develop novel end-host protection schemes. *These are well-known hard problems for which the PI will research efficient solutions.* The PI proposes the following specific tasks:

Task T1: Automatically Find and Prioritize Bugs Based Upon Their Exploitability. This research proposes to develop automatic exploit generation techniques that can be used to prioritize bug reports by their exploitability, and to find previously unknown exploitable bugs in software. The central principle is *if we can automatically construct an exploit, then so can a real attacker.* Such bugs should be fixed first. For example, bugs that allow an attacker to execute arbitrary code should be fixed before bugs that simply crash the program, and both should be fixed before bugs that do not confer an advantage to the attacker.

Task T2: Safe Patch Distribution and Application. Simply developing a patch for a bug is insufficient; defenders must distribute the patch safely to all vulnerable users. Preliminary work by the PI has shown attackers can analyze a patch to reverse engineer the bug and automatically generate an exploit in as little as 5-29 seconds. This is called *automatic patch-based exploit generation* (APEG). Our results imply that current patch distribution architectures, such as Microsoft Automatic Update, can help attackers because they distribute patches over days. This research proposes new methods for secure patch distribution, and in particular, methods for defending against APEG. Second, the PI proposes research on client-side defenses based upon the patch. The most obvious defense is to install the patch. Unfortunately, users shun patches out of fear the patch will break their system. The PI will develop alternatives to patching where patches are analyzed and used to create filters that remove exploits from the input stream, as well as research techniques for safe patch application.

Task T3: Binary Analysis Techniques. Tasks 1 and 2 require the ability to analyze binary (i.e., executable) code. When determining the exploitability of a bug, low-level details such as the stack layout, what compiler optimizations were used, and the exact run-time semantics of an instruction may all matter. In order to create safe patch distribution architectures defenders need to understand the limits of reverse engineering and APEG. For patch application, defenders need to be able to analyze the code that end-users receive to see if it will break their systems. The proposed research focuses on (1) creating a reusable architecture which we will make available in source form to other researchers, (2) performing efficient symbolic execution, and (3) performing efficient combined static and dynamic analysis.

Broader Impact: This research will develop new theoretical models, techniques, and efficient implementations for bug prioritization, patch distribution, and patch application. These results are applicable to the broader scope of software development in general. The specific program analysis techniques will be of interest to related domains, such as model checking, compilers, and formal methods. In addition to the academic broader impact, the proposed work is of interest to the security industry and to the government, and also has applications to offensive computing. The PI will make our tools available to other researchers. The PI has included letters of collaboration with UC Santa Barbara, Western Michigan University, Lockheed-Martin, and Symantec.

The PI is developing a website that offers complete software security courses, including lecture notes with integrated labs. This material will be made freely available. The PI is also developing a software security lab for eduction that will provide a safe environment for students to study attacks and defenses. Finally, the PI is engaged in outreach activities, including an annual two-week program for developing security courses for historically Black and Hispanic-serving universities [153].

CAREER: Towards Identifying and Eliminating Exploitable Software Bugs

1 Introduction

Buggy programs are one of the leading causes of hacked computers. While trojans, viruses, worms, and distributed denial of service attacks are all currently considered some of the most serious threats to computers and networks [7], all typically require that the attacker first exploit a buggy program in order to break into computers. The most direct way, therefore, to make computers more secure against trojans, viruses, and similar problems is to fix bugs before attackers can exploit them.

Unfortunately, bugs are plentiful. For example, the Ubuntu Linux bug management database currently lists over 58,000 open bugs [56]. Specific widely-used programs such as Apache 2.0, Firefox 3.0, and the current Linux 2.6 Kernel have respectively 544 [53], 501 [54], and 1337 [55] open, non-trivial bugs. Closed-source projects are likely to have similar statistics. Those are just the bugs we know about; there is also a persistent threat of zero-day exploits where attackers discover and craft an attack for previously unknown bugs. Thus, the question is not whether an attacker can find a security-critical bug, but *which bug the attacker will find and exploit first.*

Therefore, we need techniques that find bugs and prioritize bugs by their exploitability. Bugs that allow an attacker to execute arbitrary code should be fixed before bugs that simply crash the program, and both should be fixed before bugs that do not confer an advantage to the attacker. Such techniques will likely always be useful, as we do not yet know how to reliably write bug-free programs. More simply: since we cannot write secure software from the beginning, we should at least guarantee that security will improve as the software evolves.

My central research goal is to develop methods for minimizing the ***attack window*** *— the time from when an exploitable bug is introduced to when the software patch is applied on the end-system.* In this proposal, we identify several key components that significantly contribute to this vision.

Proposal Overview. We set out a research agenda for developing techniques, attack models, and theoretical foundations for minimizing the attack window. We look at the entire window, from when a bug is discovered to when a user eventually applies the patch. The core activities in our proposed research are: (1) identify and prioritize bugs by their exploitability such that they can be fixed first, and (2) develop techniques for distributing and safely managing patches for exploitable bugs. ***Our research involves collaborations with Lockheed Martin, Symantec, Western Michigan University, and UC Santa Barbara.***

Our research is motivated by several deceptively simple to state yet difficult to answer questions. Out of the 58,000 open Ubuntu bugs, which should be fixed first? Which bugs are exploitable? Can an exploit execute arbitrary code, steal information, or only crash the program? Can we find previously unknown bugs? How do we safely distribute the patch to all users? How can we get users to install the patch? Can we offer protection even if users do not install a patch immediately? In essence, how do we minimize the attack window?

Challenges. There are three central challenges to minimizing the attack window:

A. Source code analysis alone is insufficient and inadequate. Source code analysis reports errors with respect to the abstractions in the source language. A source code error, however, does not describe how the bug may be exploited as these semantics fall outside the scope of typical source languages. Further, there may be many different *classes of exploits*, e.g., arbitrary code execution vs. private information theft, that are possible. Determining which exploit classes are possible often depends upon details of the binary (executable) code, such as the exact heap layout, the stack layout, compile-time optimizations that may have been performed, and potentially even timing behavior of instructions. Further, bugs in the source language may not correspond to exploitable bugs in the compiled code (i.e., *false positives*), and may miss exploitable bugs in the compiled code (i.e., *false negatives*). We give three examples in Figure 1, which we also use throughout this proposal to illustrate ideas. *Note that we do not restrict ourselves to problems in type-unsafe languages; it is possible to write programs that have exploitable bugs in any language.*

1. *Exploit classes.* Figure 1a shows a program assumed to be written in C that is vulnerable to an integer overflow. This example is motivated by a similar bug in Internet Explorer (IE) 6 [139]. We assume `i` is a 32-bit integer, and therefore all arithmetic is performed modulo 2^{32}. On line 5, the programmer allocates s bytes of memory, and then copies i bytes of data into it on line 6. An integer overflow can occur on lines 3 and 4 when $i \geq 2^{32} - 3$,

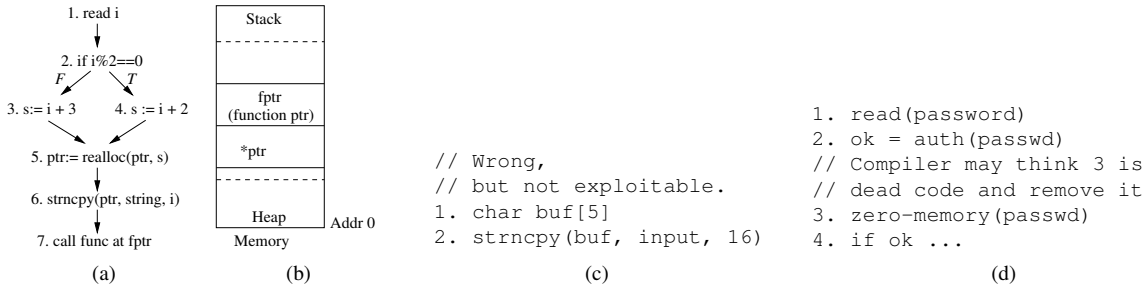

Figure 1: (a) is an integer overflow bug, with assumed memory layout shown in (b). (c) is not an exploitable bug. (d) is a compiler-induced bug that cannot be found via source code analysis.

resulting in $s < i$. While source code analysis can find this bug, it cannot show how it may be exploited. In Figure 1b, we show the memory layout of the program. In the simplest case an attacker will write outside the memory allocated to `ptr` such that the program crashes. In the worst case an attacker can overwrite the function pointer (`fptr`) with the address of code of their choosing, which will then be executed by line 7. In order to demonstrate that both cases are possible, we must analyze the binary (executable) code.

2. *False positives.* Bug-finding tools often empirically have false positive rates from 30-100% due to imprecision in the analysis itself [62, 66, 93, 158]. For example, one of the bug reports in Engler *et al.* [62] found 83 real bugs, but also reported an additional 260 false positives [62]. In addition, source-level bugs may not correspond to any exploitable condition. Figure 1c shows a C program that is buggy at the source level, but *is not exploitable* on many modern systems. Compilers such as gcc 3.3 for x86 will align memory accesses, which has the effect of allocating 16 bytes for `buf`, not the requested 5. Therefore the `strncpy` procedure, which copies 16 bytes of `input` into `buf`, is incorrect (and should be fixed eventually), but is not exploitable.
3. *False negatives.* Source code analysis may miss compiler-induced security-critical bugs [44, 81], object-reuse bugs [4, 86], code that uses implementation-specific aspects of the programming language (e.g.,[64, 151]), and bugs dependent upon the timing characteristics of hardware (e.g., [19, 91]). For example, Figure 1d shows an authentication routine where the programmer zeroes out the password from system memory in order to limit the lifetime in main memory, which is used to mitigate a variety of attacks [4, 40, 76, 81, 86]. Unfortunately, many modern compilers will optimize away the code on line 3 since the zeroed `passwd` is never subsequently used (thus is considered dead code).

B. Everyone should be able to audit the code they run for security-critical bugs. There is no doubt that source code analysis is a powerful tool for finding security-critical bugs. Nonetheless, the lack of access to source code should not impair a user's ability to audit the code they run on their own computers for exploitable bugs. Further, we observe that most current attackers typically do not have access to source code, yet are effective at finding and exploiting security-critical bugs. We want to understand both the limitations and possible advantages of a binary-only approach in order to accurately model and replicate the capabilities of attackers.

C. Patch creation alone is insufficient. We must consider all the steps necessary to get patches installed on vulnerable systems, including patch distribution and patch application. The PI showed that attackers can use patches to *automatically* reverse engineer the bug that is fixed and *automatically* generate an exploit [28]. After generating an exploit from a patch, attackers can then launch a *delayed patch attack* against all the users who have not yet received or applied the patch. **We generated an exploit in as little as 5-30 seconds** (see Section 3 for details). Our results imply that attackers should be considered armed with an exploit seconds after the patch is public. Therefore, current patch distribution architectures, such as Microsoft Windows Update, are insecure since most users will not have downloaded a patch within the time-frame an attacker could automatically generate an exploit and attack their system.

1.1 Overview of Proposed Work and Technical Contributions

Our proposed research and tasks consist of two underlying thrusts. First, we use binary program analysis in techniques that minimize the attack window. Second, we propose new program analysis directions for analyzing and formally reasoning about binary code. These two thrusts are synergistic: improvements in binary analysis allow us to get better results for our applications, and our applications guide the creation of successful and useful program analysis (as well as serve as a metric of success). The specific tasks are:

Task T1: Automatically Find and Prioritize Bugs Based Upon Their Exploitability. We first investigate the capabilities of an attacker. First, given a list of bugs, can we (a) automatically generate exploits in order to prove a bug is exploitable, and (b) determine the different classes of exploits that are possible (e.g., steal information, execute arbitrary code, etc.). *If we can automatically construct an exploit, then so can a real attacker.* Such bugs should be fixed first. For example, we would like to show which of the 58,000 Ubuntu bugs can be exploited, and automatically generate representative exploits.

Second, we explore finding novel bugs using binary analysis. For example, can we find typical bugs such as the integer overflow in Figure 1a? Can we find bugs that are not typically found by source code analysis, like Figure 1d? The overall contribution of this task is to develop foundations for determining the exploitability of a bug, as well as specific techniques that are of interest to industry, e.g., for bug management and offensive computing.

Task T2: Safe Patch Distribution and Application. The main contribution of Task 2 is to minimize the attack window once a patch has been developed. First, we will research the requirements and solutions for a secure patch distribution architecture. The main goal is to prevent patches from helping attackers. Note this problem is (perhaps unintuitively) non-trivial. For example, three possible solutions are to use obfuscation, use a fast patch distribution architecture such as P2P, and use encryption. Our preliminary research indicates that naive implementations of these approaches have limitations (see Section 4). We propose new directions for a realistic and secure patch distribution architecture.

Second, we propose research on client-side defenses based upon the patch. The most obvious defense is to install the patch. Unfortunately, users shun patches out of fear the patch will break their system. A vast number of major security incidents such as Conficker [149], Code Red [38], Blaster [148], and Slammer [107] exploited bugs *for which there was already a patch available that users simply did not install.* We will develop alternatives to patching where patches are analyzed and used to create filters that remove exploits from the input stream. In the long term, we will also research techniques for safe patch application.

Task T3: Develop Efficient Binary Analysis Techniques. Our tasks require that we be able to faithfully reason about binary code. The PI has previously developed an initial prototype binary analysis toolkit called Vine [17]. Guided by lessons learned by our first-generation work, we will build the next-generation binary analysis platform (BAP). There is a widespread need for such binary analysis tools in the public, private, and research sectors, as indicated in part by our 5 letters of collaboration. We will make the BAP source code available to other researchers.

Two specific binary analyses techniques we will develop for this project are efficient symbolic execution and combined static and dynamic analysis. These techniques will improve the efficiency of our core approaches for Tasks T1 and T2. Previously, the PI developed an efficient backward analysis (weakest preconditions) for binary code [30]. However, there is significant interest in techniques that work in the forward direction (forward symbolic execution), e.g., [13, 22, 33, 35, 36, 45, 46, 48, 73–75, 90, 94, 109, 140]. Unfortunately, these forward techniques are exponentially more expensive than the backwards techniques. We have developed an algorithm and proof of correctness for the forward-style direction that achieves the better theoretic guarantees offered by our backward analysis. We will implement the algorithm in BAP and determine if the theoretic guarantees translate into real-world benefits for our applications and those that are using the forward-style analysis. Second, we will develop and implement extensions to BAP for performing a combination of static and dynamic analysis.

2 PI's Prior Research Accomplishments

The proposed project integrates research from software security, program analysis, applied cryptography, and network security. The PI has extensive knowledge and research experience in these areas, and has made several key contributions. *Three of the relevant papers were recognized as conference "Best Papers" [18, 20, 26].*

Software Security and Program Analysis. Software security and program analysis are important for finding bugs and prioritizing them (Task 1), designing analysis-resistant patch distribution schemes (Task 2), and efficient binary analysis (Task 3). For example, the work on efficient forward symbolic execution in Task 3 builds upon the PI's expertise in developing efficient weakest preconditions for unstructured code [30], the initial algorithms for prioritizing bugs for Task 1 will build upon the PI's work in [28], and the PI's experience analyzing malicious code [22, 23] is invaluable in designing analysis-resistant patch-distribution architectures (Task 2).

The PI's PhD thesis was on "The Analysis and Defense of Vulnerabilities in Binary Code" [17]. In the thesis, the PI proposed novel techniques for (1) automatically generating filters for an intrusion detection system by performing program analysis of the vulnerability [26, 27, 30, 120], (2) reverse engineering a software patch to develop an exploit, which we call automatic patch-based exploit generation (APEG) [28], and (3) efficient binary analysis and efficient algorithms for computing the weakest precondition on unstructured code, which reduced the time and resulting verification conditional size from exponential to quadratic [30]. The work on filter generation builds the first formal foundation for arguing about filter accuracy (i.e., how well it filters exploits while not filtering safe inputs). The work on filter generation [26, 27, 30] has been used by Symantec to improve their desktop product called Symantec 360 [16], and has been subsequently licensed to Ensighta, Inc., and is used in a joint contract with Reservoir Labs.

As part of his thesis, the PI built *Vine*, a prototype static binary analysis infrastructure. Vine is the static analysis component of the BitBlaze project [3]. The PI has used *Vine* with other software security research such as automatically finding protocol implementation bugs and inconsistencies [20], and automatically reverse engineering and dissecting malicious binaries (malware) [22, 23]. The PI has also used Vine for program analysis research, such as a new alias analysis for binary code [25]. *Vine* is currently used by about a dozen different research projects at universities such as UC Berkeley, the University of Pittsburgh, UC Santa Barbara, as well as at companies such as Lockheed-Martin, Symantec, and Ensighta.

The PI also has made novel contributions in source code analysis, including automatically partitioning programs for security [29] and automatically protecting programs against integer-based vulnerabilities [21].

Applied Cryptography. At least two aspects of this project benefit from the PI's experience in applied cryptography: a) finding side-channel bugs that allow attackers to break cryptography (see Task 1.2), and b) secure patch distribution (Task 2). The PI has developed the first successful remote timing attack against RSA as implemented in real software. In particular, the PI showed that one could break a 1024-bit RSA key in an OpenSSL-enabled webserver in about 2 hours across the network [18, 19]. Major RSA implementations cite the PI's work as the reason they now adopt defenses (e.g., [79, 122, 154]).

Network Security. The PI has made contributions in the areas of network worm analysis [24], network-based defenses against fast spreading attacks [117, 152], network protocol analysis [14, 116], and secure network deployment [96, 136]. The PI's work in worm modeling [24] is related to the problem of efficient patch distribution (Task 2). The expertise in protocol analysis [14, 116] is important for designing secure patch distribution protocols. One of the PI's work in network protocol analysis [14] is patent-pending in collaboration with Microsoft. The PI has proposed using network graph analysis to optimize and secure virtual machine deployment, configuration, and migration [96, 136]. The PI's work on network-wide deployment of virtual appliances was granted a patent in May, 2008 [96]. The patent has been licensed to Moka5, Inc. and is an integral part of their business.

3 Task T1: Automatically Finding and Prioritizing Bugs Based On Their Exploitability

Buggy software is inevitable. All bugs, however, are not equal. Bugs that allow attackers to steal private information, cause denial of service, and hijack control of a user's computer should be considered more critical than bugs that do not. How do we *automatically* tell which bugs are likely to be exploited from those that are not? Can users without access to the source code audit their code for exploitable bugs? In this task, we describe our research agenda for prioritizing known bugs, which we will extend in the latter part of this task to finding novel exploitable bugs.

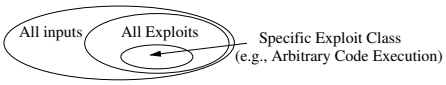

Figure 2: We view an exploit class as a subset of all exploits, which is a subset of the program input domain.

3.1 Task T1.1 (Years 1-3): Prioritizing a List of Bug Reports

Problem Description. Informally, a program's input domain can be partitioned into the set of safe inputs for which the program operates as intended, and the set of exploits. Within the set of exploits there are particular exploit classes, such as crashing the program, executing arbitrary code, and information theft. Figure 2 depicts this intuition. For example, in Figure 1a, the input domain is 32-bit integers, the set of all exploits are the inputs $i = \{2^{32} - 3, 2^{32} - 2, 2^{32} - 1\}$ and all possible inputs for `string`. The set of exploits that execute arbitrary code are the same except `string` must overwrite the function pointer `fptr` with the address of the code to execute.

Our goal is to prioritize bug reports by their exploitability. Specifically, for each bug report we want to determine:

1. Whether the bug is exploitable. In our setting, we take a (logically) *sound* approach: if we say a bug is exploitable, it really is exploitable. The flip side of soundness is completeness, which in our case would mean that we would find all bugs which are exploitable in a list of bugs. Unfortunately there is no algorithm that is both sound and complete for determining the exploitability of (important classes of) security-critical bugs [80, 97]. Our soundness choice means we will not be able to always produce an exploit when one exists. This limitation is outweighed by the advantage that we will never be fooled by a spurious bug reports, and therefore we will never misdirect attention to an unexploitable bug. The main technical challenge is to show that the buggy line(s) of code are indeed reachable, and generate an example input that causes an unsafe execution. Note this question is closely related to model checking, automatic test case generation, and similar fields.
2. Specific classes of possible exploits. There can be many different kinds of exploits for a single bug. For example, security professionals typically differentiate between crashing the program and permitting arbitrary code execution; the latter being much more serious than the former. More formally, an exploit class is defined by a predicate on the space of possible bad executions. Central challenges and questions in this problem include (a) what are the relevant exploit classes? (b) what environmental factors, e.g., program configuration, affect the possible set of exploits and how can we model them?
3. A witness exploit for each exploit class. A witness exploit acts as a proof of soundness: if the witness does indeed exploit the program, the bug is real. Witness exploits are not just for show: witnesses help programmers write patches by giving them a specific instance that they can use to debug. Research questions include what kind of witness exploits can we generate? How efficiently can we generate them?

Preliminary Work. The PI's PhD thesis established preliminary techniques for automatically generating exploits once a patch has become available. This work is an initial step in developing the theory and practice of automated exploit generation in general. Note that our preliminary work did *not* automatically prioritize bugs or automatically determine the types of exploits possible.

In our preliminary work we focused on input validation bugs, which are considered the most dangerous programming error by computer security experts [132]. We say a bug is an input validation bug if exploits are detectable by an execution monitor [17, 138]. Most typical bugs fall under this definition, e.g., stack overflows, heap overflows, leaking private information, etc. Inputs where the monitor would abort the program are considered exploits, e.g., the "All Exploits" subset of Figure 2. Note that this definition does not explicitly distinguish between exploit classes, e.g., an exploit that crashes Figure 1a is considered equivalent to one that allows arbitrary code execution; we handle this case below.

The *patch-based exploit generation* problem is: given a buggy program B and a patched version of the program P, generate an exploit for the potentially unknown vulnerability present in B but fixed in P. The PI was the first to show **automatic** patch-based exploit generation is possible against input validation bugs. The intuition was that patches for such bugs typically introduce new checks that weed out exploits. The new checks in P reveal (a) where the bug is in the code, and (b) what conditions trigger the bug. For example, one fix to the integer overflow in Figure 1a is to add a new check before line 5: `if` $s < i$ `goto error else goto 5`. Previous approaches to exploit generation could automatically find such added checks, but could not automatically generate inputs that fail the new check.

Our approach to automatic patch-based exploit generation (APEG) consists of four steps:

1. Find the new checks added to P by differencing the two programs. There are many off-the-shelf utilities to perform syntactic differences at the binary (e.g., [60, 135]) and source level (e.g., the standard Linux `diff` command). Diffing would return the new checks, e.g., `if` $s < i$ `goto error else goto 5`.
2. Generate an input that executes the new code path introduced by the check. In the example, we would generate an input that executes line A and fails the check. We calculated the weakest precondition on the input domain to fail the new check. The result is a formula that is true for all inputs that would fail the check and execute the new branch in P:
$$((i\%2) == 0 \Rightarrow (i + 2\%2^{32}) \leq i) \wedge ((i\%2) \neq 0 \Rightarrow (i + 3\%2^{32}) \leq i) \quad (1)$$
3. Solve the formula, e.g., $2^{32} - 2$ is one satisfying answer. The satisfying answer is a *candidate exploit*.
4. Verify the candidate exploit x is a real exploit by running $B(x)$. We assume we are given a run-time monitor that can verify that execution is exploited (e.g., stack-protection aborts the program).

We tested our approach with 12 Windows vulnerabilities that have been patched. APEG succeeded in 5 cases. The fastest end-to-end time for generating an exploit was 29 seconds when we performed step (1); time was reduced to 5 seconds if the diff was given to us. The longest it took to generate an initial proof-of-concept exploit was 456 seconds. Thus, we conclude automatic patch-based exploit generation is possible.

Proposed Work and Approach. Our preliminary work demonstrates that automatic exploit generation is possible. It does not show that we can automatically use it to prioritize bugs, and automatically generate exploits in different classes. At a high-level, our proposed work consists of the following steps that build upon our preliminary work: (1) formalize the bug report as a predicate on the program state space, (2) generate a formula that captures the conditions to reach the buggy line of code and solve the formula (e.g., using a decision procedure) to determine if the bug is exploitable at all (this formula describes all exploits in Figure 2), (3) refine the formula to specific exploit classes, and solve again for specific exploits. Steps (1) and (3) are new in the overall approach, which we discuss at more length below. We discuss efficiency improvements to step (2) in Task 3. We also discuss challenges that our preliminary APEG work flushed out, and the proposed research directions.

Step 1: Precise Bug Reports. Previously, we were provided with a tight specification of the bug via the patch. Bug reports, however, typically do not state what is wrong with such precision. For example, a bug report may simply state that lines 3 and 4 of Figure 1a are buggy, but not tell us why. How big of a limitation is it when the bug report is imprecise? How much imprecision can we tolerate? Our current approach focuses on integrating our system with open-source bug finding utilities such that a specification of the safety property thought to be violated is produced along with the bug report. In the future, we expect that we can often infer a precise bug specification from an imprecise bug report. For example, in Figure 1a we can infer that since the line is an integer operation, overflow will occur when $s \leq i$.

Step 3: Generating Specific Exploits. The main intuition behind generating specific exploits in step 5 is that such classes are refinements on the formula. For example, the safety property for Figure 1a says that overflow should not occur. The class of exploits that lead to arbitrary code execution can be specified as overflow should not occur *and* `fptr` should be overwritten with a value that points to the code to execute.

We have successfully manually tested out adding such refinements to the predicate in [28]. The goal is to completely automate this step. Our plan is to include refinement predicates as template code which can be checked after an initial proof-of-concept is generated. For example, we could refine the formula to include logic that says "the input should overwrite the return address with a value that will execute code x", where x is the location of the known stored return address for buffer overflows.

In the long term, we would like to explore both building a large library of such templates, as well as explore methods for automatically generating them. We also want to explore techniques for narrowing down the set of specific exploit classes possible without an explicit call to the decision procedure, since that step is typically the most expensive.

Challenge: Can we better reason about program loops? Our preliminary efforts did not attempt to reason about loops. We simply unrolled them a fixed number of times. Although unrolling loops is common (e.g., in bounded model checking [12]), it prevented us from automatically generating exploits when one was possible in some cases. We have begun to implement structural analysis, induction variable analysis, and other typical compiler analysis [5, 110] to

find common loop patterns and bounds. We have also worked with U. Pittsburgh researchers for developing initial loop-driven analysis [167], and plan to investigate more advanced template-driven analysis [78] and loop-extended symbolic execution [137].

Challenge: can we create exploits for loop-dependent bugs? More concretely, our previous work did not work well when exploits are loop-dependent. Unfortunately, there are a large class of bugs and exploits that depend upon loops, with typical `strcpy` and similar out-of-bounds writes being at the top of the list. For example, the typical buffer overflow in C is:

```
void copy(char *dst, char *src)  while(*src != NULL) *dst = *src; src++; dst++;
```

The `strcpy` library function, for instances, is implemented as above on many Linux systems. This code is exploitable when the string length pointed to by `src` is longer than `dst`. The loop analysis discussed above will allow us to better handle these sorts of bugs. The longer-term questions we look to address with experiments include: Which loops can we easily reason about in order to generate exploits? Are there other heuristics that work well? Is there a better way to reason about strings and string manipulation loops?

Challenge: Better Automation and Distributed Implementation. Our initial work on APEG had three separate components: differencing, candidate exploit generation, and exploit verification. We did not fully automate the interactions between these components. Adding automation is simply an engineering challenge, thus was omitted during our initial preliminary research. Nonetheless, we need complete automation in order to work in the proposed setting where we want to prioritize thousands of bugs. Further, our implementation did not take advantage of the fact that many operations could be parallelized or distributed.

Challenge: Larger Classes of Bugs. Initially we will continue to focus on input validation bugs. As the project matures, we will expand our scope to look at other classes of bugs. In particular, we would like to look at exploit generation for the larger class of safety properties (not just those enforceable by an execution monitor [138]), time-of-use to time-of-check bugs, cryptographic timing attacks, and others [86, 88].

Evaluation Metric. Our research has a built-in evaluation metric: the percentage of exploitable bugs for which we can automatically generate an exploit. Our research plan, therefore, includes testing against known exploitable bug data sets. Our collaborations with Lockheed-Martin and Symantec will help secure such data sets. We also plan on testing our techniques on lists of bugs which have potentially unknown exploitable bugs, e.g., the Ubuntu bug database.

3.2 Task T1.2 (Years 3-5): Finding New Exploitable Bugs

Problem Overview and Definition. In Task T1.1, we assume that bug-finding is done for us, e.g., via a bug-finding tool. In this task we will investigate two directions for auditing code for new bugs. First, we propose to develop static analysis tools and techniques for finding bugs in binary code. Second, we propose a combination of source and binary level analysis to find bugs that cannot easily be detected at either level alone. Task 1.1 and 1.2 together yield an end-to-end approach for finding and prioritizing exploitable bugs.

Preliminary Work. The PI has developed preliminary methods for finding bugs and inconsistencies in network protocol implementations based upon binary code analysis [20]. In this problem we are given two binary implementations P_1 and P_2 of the same network protocol specification, e.g., two different web-servers that implement HTTP. Given the same configuration, we say P_1 and P_2 *deviate* on input i if $P_1(i) \neq P_2(i)$. Deviations are important to security because they indicate possible bugs, and they also allow an attacker to identify specific protocol implementations. Automatically finding deviations, however, is challenging because we would expect that $P_1(i) = P_2(i)$ for most i, as both implement the same protocol. The PI developed an algorithm for automatically finding such deviations by: (1) symbolically executing P_1 and P_2 on a common input i to produce formulas f_1 and f_2, respectively, and (2) creating a candidate deviation by solving for x such that $f_1(x) \neq f_2(x)$, and (3) verifying the candidate deviation is a real deviation. The main idea is in step (2). If $\pi_1(i), \pi_2(i)$ are the code paths executed by $P_1(i), P_2(i)$ respectively, then candidate deviations are x's such that $\pi_1(x) = \pi_1(i)$, but $\pi_2(x) \neq \pi_2(i)$.

We found several deviations in NTP and HTTP implementations. For example, Apache (P_1) and Mini-web (P_2) both implement HTTP. We chose i to be `GET /index.html`. Apache and Mini-web returned the same web-page. Our techniques then generated the deviation `GET .index.html` in about 35 seconds. Tests confirmed that Apache will correctly return a "file not found" error when the page does not exist, while Mini-web returns `/index.html`. The reason was Mini-web always assumes the first character of a URI is "/" for performance reasons.

Proposed Work and Approach. Our preliminary work developed techniques that take a dynamic software model-checking approach. Such techniques are fairly heavyweight. Static analysis, however, is a complementary lightweight approach that can also find a wide variety of bugs [63].

Can we find a comparable number of bugs at the binary level as at the source code level using similar techniques? We will design and adapt bug-finding techniques typically used at the source level to answer this question, which will (a) give security auditors another tool for analyzing the security implications of software, and (b) help us understand to what extent attackers are limited by only having access to binary code.

The first analysis we plan on implementing is static taint analysis, which looks for unsafe uses of user input [66–68]. Static taint analysis is promising for two reasons. First, many exploitable bugs we have seen could potentially be detected with a taint-style analysis. For example, a significant number of the 30 web-browser bugs in [2] are amenable to such analysis. This concurs with previous work using taint analysis on source code (e.g., [67, 85]). Second, dynamic taint analysis has previously proven effective at the binary level,e.g., [121, 147, 164]. Dynamic analysis, however, is limited to a single path; a static approach could cover many paths. We also have preliminary results for finding known buggy code that is reused in new software components, similar to [69, 101]. We will continue to implement new analyses that are informed by source-level bug finding techniques to explore how well they work at the binary level.

Can we find exploitable bugs that are not traditionally found with source code alone? We are particularly interested in compiler-induced bugs, such as in Figure 1d. One method for doing this is to check for the desired security guarantees at both the source and binary levels. This is similar in spirit to work such as translation validation [113, 129, 168]. One difference with previous work in the area is we want to verify the preservation of a security property instead of verifying the correctness of an optimization. We are also interested in automatically finding side-channel attacks in software. Despite their importance, there has been little work on automatically finding such attacks in software. Our experience (e.g., from [19]) indicates that timing attacks usually arise from control dependencies between user input and the secret key, which static analysis may be able to find. Intuitively, different code branches execute depending upon bits in the key. When one branch takes longer than the other, the overall timing reveals information about the bit.

4 Task T2: Safe and Secure Patch Distribution and Installation Techniques

In Task 1, we identified exploitable bugs so that they can be fixed by the developer via a software patch. The attack window is not closed from the perspective of the user, however, until they receive the patch and the appropriate patch-based defenses take effect. In this task we propose research directions to minimize the attack window once the patch is available by securely distributing the patch, and developing safe-to-install patch-based defenses for end-hosts.

4.1 Task T2.1 (Years 1-3): Secure Patch Distribution

Problem Description. At first glance, releasing a patch that addresses an exploitable bug can only help security. We must, however, take into consideration the entire time line of patch distribution. Current automatic patch distribution architectures, such as Windows Automatic Update, stagger patch roll outs over days [71]. Our work on APEG shows an attacker can create an exploit in seconds. Note that the patch tells the attacker about a bug he may not have known about previously. Also note that most vendors, including Microsoft, disclose whether a patch is security-related or not.

We therefore conclude that current patch distribution architectures, such as Windows Automatic Update, can hurt security because they allow *delayed patch attacks*. In a delayed patch attack the first user to get a patch can use the patch to create attacks against users who have not received the patch. Thus, those that first get a patch are at a significant security advantage.

We observe that the delayed patch attack is not specific to Internet updates: it is universal to any software system that requires updating. For example, tanks, ships, trains, and airplanes all have sophisticated computers which may be networked in order to provide advanced situational awareness. If an attacker can capture a pre-patch and post-patch version, they can potentially attack all the remaining unpatched vehicles. Solutions we develop to the delayed patch attack will therefore be applicable in any setting when APEG is possible.

Problem Definition. Our primary research question is *can we develop practical methods that prevent delayed patch attacks?* We had originally thought the problem was easy; we were wrong. Considered the following possible solutions:

1. Fast Patch Distribution. One idea is to try to deliver patches quickly to end-users, e.g., via a P2P network. Current patches are typically whole new programs that are megabytes in size. When patches are complete programs, APEG succeeded in as little as 29 seconds. Suppose vendors distributed patch deltas that only included the changed lines of code, e.g., as in [58, 126]. Such deltas would be smaller, thus take less time to distribute. Access to deltas, however, drops APEG time to only about 5 seconds [28]. Preliminary work by Microsoft researchers indicate that even a P2P network may not be able to get patches out that quickly [28].

2. Patch Obfuscation. Vendors could obfuscate patches to prevent attackers from learning what has changed between the buggy and patched program. One advantage to using obfuscation is it would not require changes to the distribution architecture itself. There are, however, several potential problems. First, obfuscation tends to slow down programs. *Thus, the patched version would run **slower** than the original program.* Users are likely to resist patching if it is guaranteed to slow down their computers. Second, there are known theoretical results that show perfect obfuscation is impossible in a black-box model [11]. Finally, although obfuscation is used in practice, (e.g., [43, 105, 150]), there is also recent work that successfully breaks some of these tools (e.g., [70, 83, 95, 141, 155]).

3. Encryption. One could initially encrypt patches so that they can be distributed without leaking information about the bug [28, 84, 133]. Then, after a suitable time period, a decryption key is broadcast. When users receive the key, they decrypt the patch and apply it. One can argue this scheme is fair: everyone has the opportunity to apply the patch at the same time. There are, however, several potential problems. First, even though the key is small, we still encounter the problem of having to get it to everyone simultaneously. Second, it is unclear *when* the key should be distributed, e.g., how do you know when enough people have received the patch such that releasing the key is safe? Third, this approach only provides protection for people who apply the patch immediately (thus no patch testing!), which is not a common practice in business.

Proposed Work and Approach. We propose to develop a realistic patch distribution scheme that defends against delayed patch attacks.

Straightforward First Steps. Our first step is to solve one immediate problem: computers that have been offline should consider themselves not up-to-date and vulnerable to delayed patch attacks. For example, if a computer is reinstalled from the manufacturers source (e.g., reinstall XP from DVD in order to remove malware), the system will be out-of-date and vulnerable when it first comes online and will likely be compromised [37]. We plan on implementing a daemon for Linux, Windows, and OS X that will initially limit network access when reconnecting to the Internet to only allow the system to check for updates. We will make the daemon freely available. We expect this to be a straightforward important step to eliminate the danger posed by delay patch attacks to offline hosts.

Next Steps: Multi-Stage Patch Distribution. We plan to develop approaches that do not suffer the drawbacks listed above. Recall the main drawbacks to obfuscation is it only provides temporary protection and it may slow down programs. Our next steps are to research a multi-stage patch distribution approach. In particular, we are investigating an approach where we initially distribute an obfuscated version, and then at a later time release an unobfuscated version. In this scheme temporary protection is sufficient; we just need to prevent delayed patch attacks until everyone gets the patch. The unobfuscated version limits the impact of any performance degradation initially experienced with the obfuscated version.

Our plan requires addressing three issues: obfuscation, fast dissemination, and proving equivalence between the obfuscated and unobfuscated version. First, can we estimate the amount of time it will take to deobfuscate a program? What are the knobs we can turn to make obfuscation more difficult to crack in practice? Can we design obfuscation schemes that are about as efficient than optimized code?

Second, we still want fast patch distribution since the faster we can distribute the obfuscated version the faster we can subsequently release the unobfuscated version. Microsoft researchers have performed initial studies on patch distribution schemes, including the theoretical limits [71, 157]. Empirical observations show strong time-of-day effects across the globe [71]. We should be able to turn these models into optimal strategies for patch distributions. We plan on partnering with industry affiliates of the PI's security lab (CyLab [52]) in order to test out such strategies.

Third, we want to convince users that the unobfuscated and obfuscated versions are equivalent, e.g., so that they know any testing performed on the obfuscated version is relevant to the unobfuscated version. One approach is by showing semantic equivalence, which is relatively heavy-weight. We hope to develop key-based obfuscation schemes where the deobfuscated version is derived automatically from the obfuscated version with a key. This is related to DRM, but not the same since the goal in DRM is to hide a decryption key in the program, not have key-based

deobfuscation. Are such schemes practical? Is proving state equivalence in any approach sufficient, since it does not cover effects such as timing?

4.2 Task T2.2 (Years 3-5): Safe Patch Installation

Problem Description. The more quickly users install a patch once they receive it, the smaller the attack window. Unfortunately, users do not like to install patches immediately. One significant reason is people fear patches will break their system. We propose (a) patch-based defenses that do not require patch application, and (b) long-term research for making patches safer to apply on end hosts.

Proposed Work and Approach. While users often do not install patches immediately, users (and businesses) typically do use intrusion prevention products (e.g., Snort [130] and Symantec 360). Such systems use *filters* that recognize and discard exploits from the input stream. [1] A filter differs from a patch in at least two ways: (1) filters are interposed on the input stream between the buggy program and the user, and (2) filters typically discard exploits, while patches would take a semantically meaningful action.

While users should eventually install a patch, our immediate direction is to leverage IDS systems to provide temporary protection by developing mechanisms to *automatically generate filters from patches.* The filter would then protect the system until the patch is applied. The PI has previously developed the first theoretical models for accurate filters (i.e., not accidentally filtering safe inputs while simultaneously filtering all exploit variants) and has also implemented and tested these approaches on real bugs when only binary code and a single sample exploit are available [17, 27].

The most straightforward approach of our previous work [27] would first difference B and P to find new security checks, and then generate a filter that weeds out exploits that fail the new checks. Unfortunately, we do not want to make it easy to reverse engineer the exact bug fixed by the patch due to delayed patch attacks. How can we resolve this tension? Further, our previous work was designed to interact with existing intrusion detection/prevention systems for networked servers. Filters work well in such environments because there is almost always a natural place to interpose a filter check, e.g., where the server reads in a request. Applying these techniques to a wider class of programs, e.g., event-driven or interactive windows programs, will require further research on the effects of interposing a filter check.

Our longer-term goal in this task is to develop mechanisms that improve the safety of patches. This is a natural extension of the initial filter-based approach; if the filter works as intended, and the filter is derived from the patch, the patch itself should be safe to apply. We plan on augmenting this work with symbolic execution mechanisms for testing that the patch has equivalent semantic behavior for safe inputs, i.e., the patch does not break previous functionality. In particular, this would be an extension of our work on deviation detection [20] where we try and show the absence of deviations on previously known good inputs. One of the challenges is to make this approach more scalable to all changes that might be made by a patch. Note that again there is a tension between revealing enough information for a user to confidently and safely apply the patch, and making it easy for attackers to generate exploits.

5 Task T3: Efficient Binary Analysis

Our proposed research directions require techniques for analyzing binary code. We need binary analysis because (a) it allows us to argue about the security of the code that will execute, and (b) it is likely to be widely applicable since everyone has access to binary code. In this task we describe our research plans for building the next-generation infrastructure, called BAP, which we will available with source code to other researchers.

Problem Description. Analyzing binary code is challenging for two reasons. First, machine instructions on typical platforms are complex to analyze. For example, in x86 there are single instruction loops (e.g., `repz`), instructions whose behavior depends on the operand (e.g., `shl`), and most instructions have implicit side effects that set up to 6 additional status flags. Consider the three line x86 program in Figure 3. Any analysis that wants to determine

[1] In IDS literature, a filter is sometimes called a "signature". Unfortunately the term "signature" is overloaded, and means something different to cryptographers.

```
// instruction dst, src
add eax, ebx
shl eax, edx
jo exploitable
```

```
if (eax + ebx > 2^32) of = 1 else of = 0
eax = eax + ebx
if ( edx is 0 or 1){
   if (top two bits of eax are the same)
         of = 0 else of = 1
}
eax = eax << edx
if (of == 1) goto exploitable
```

Figure 3: A small assembly program (left) can have complex low-level semantics (right).

when `exploitable` is reachable would be required to understand the semantics shown on the right side of the figure. Second, binary code is different than source. For example, while higher-level languages have types, functions, pointers, loops, and local variables, assembly has *no types, no functions, one globally addressed memory region, gotos and stack frames instead of local variables*. Therefore, analysis designed for a typical source code language may not be appropriate at the binary level. Our experiments and experience show that much of the higher-level semantics are irrecoverably lost during compilation.

Prior Work. We discuss other binary analysis tools in Section 7. The PI has previously built an initial binary analysis prototype called *Vine* [17]. Vine has been successfully used in about a dozen research projects at various institutions. Vine lifted x86 up to an unambiguous and explicit intermediate language, and provides a set of algorithms for building graphs (data and control flow), performing chopping [82] and slicing [162], and calculating weakest preconditions efficiently [30, 59]. Vine grew organically from a number of projects by the PI [16, 17, 20, 22, 23, 25–28, 30, 116, 120] and extensions by others [32, 33, 70, 137, 167]. Note Vine did not require debugging or symbol table information, but could take advantage of it when present.

Proposed Work and Approaches. Our research goal is to develop techniques for analyzing binary code efficiently. Our approach differs from decompilers, dynamic instrumentation, and disassemblers in that we treat binary code *as a first-class language*. At a high level, we want to faithfully analyze binary code as efficiently as source code, and provide an extensible binary analysis platform similar to source code platforms like LLVM [99], SUIF [6], and CIL [114]. In order to meet these goals, we propose developing the next-generation binary analysis platform, called *BAP*. BAP will address three central issues: it will be a general purpose architecture, implement more efficient symbolic execution techniques, and provide better support for combined static and dynamic analysis.

1. Well-Designed Architecture. Vine was not designed from the ground up as a general-purpose binary analysis architecture. Since Vine grew organically, there are few standardized API's, routines are often specially crafted for a particular project, etc. There is a widespread need for a faithful, well-designed, and extensible binary analysis architecture as evidenced by the letters of collaboration.

We have learned how an analysis infrastructure should be designed and built, and BAP will be that implementation. We have a preliminary 0.1 implementation with the core abstractions we want to provide. One of the key features of BAP is we raise up binary programs to an unambiguous, explicit, and formalized intermediate language. We have also added the ability to analyze bi-endian architectures (e.g., to analyze ARM), and now provide true 64-bit support (e.g., to analyze AMD-64).

2. Efficient Symbolic Execution. A recurring problem in program analysis is to derive a formula automatically which is satisfied by inputs that make a particular program assertion true, e.g., [13, 22, 33, 35, 36, 45, 46, 48, 73–75, 90, 94, 109, 140]. For example, we may want to know under which conditions an input to the program in Figure 3 will execute `exploitable`. In this case, the assertion would be that the zero flag should be set. Similarly, in Figure 1a we may want to know when overflow will occur, which can be asserted by $s < i$ at line 5. A main concern is generating compact formulas [65].

One method for automatically computing such formulas is to calculate the *weakest precondition* for when the assertion will hold [59]. At a high level, the weakest precondition is a *backward* computation that starts from the assertion, and inductively calculates a predicate for the assertion to hold based upon the current statement. The PI has

proved the weakest precondition for unstructured programs (such as binary code) can be calculated in $O(n^2)$ time and produces an $O(n^2)$ formula size, where n is the number of statements [30]. This proof is a generalization of [65, 100].

A second approach for generating such formulas is using *forward symbolic execution*. Forward symbolic execution executes the program on symbolic variables instead of a concrete value. Forward symbolic execution calculates a new formula for each program path, thus the final formula to reach a particular node is *exponential in time and space in the number of branches encountered.*

Although the best theoretical results show weakest preconditions are better than forward symbolic execution, forward symbolic execution is used extensively in practice, e.g., [13, 22, 33, 35, 36, 45, 46, 48, 73–75, 90, 94, 109, 140]. One reason is that we can concretely execute on the real hardware all statements that do not depend upon a symbolic input. For example, if there is a call `foo(5)`, symbolic execution can execute the call naively since the argument is a constant. The weakest precondition will treat the call abstractly, which results in the entire body of `foo` in the final formula. Thus, in practice forward symbolic execution may generate smaller predicates in some cases.

We have an initial formal proof that shows we can get the same $O(n^2)$ bound on the formula size offered by weakest preconditions in typical forward symbolic execution settings. Size is important, but size is not everything. We also want the formulas to be easy to solve [30, 65, 100]. In this task, we will (a) finish writing up the proof and verify it (e.g., via Twelf [128]), (b) build forward symbolic execution into our infrastructure, and (c) verify the smaller predicate size in typical applications, such as fuzzing and automated test case generation, are not just smaller but easier to solve. Such a result would benefit a significant number of applications that currently use forward symbolic execution [13, 22, 33, 35, 36, 45, 46, 48, 73–75, 90, 94, 109, 140].

3. Combined Dynamic and Static Analysis. We will adapt our platform and explore using a combination of static and dynamic analysis. The advantage of dynamic analysis is it provides a detailed answer to questions about a single program path (and it lets the processor do the hard work of evaluating statements). However, dynamic analysis results are only applicable to the analyzed path. Static analysis generalizes to more program paths, but will often over-approximate results. A combination of the two has proved useful in initial experiments, e.g., in APEG [28]. We will build in capabilities for combining the two in BAP.

6 Broader Impact

This research addresses known hard problems for which the PI will research efficient solutions. A successful research program for Task 1 will (a) be able to automatically create exploits to demonstrate whether a bug was security-critical, (b) determine how an attacker may exploit the bug, and (c) identify novel bugs. These steps are important for minimizing the attack window since it will help defenders find and fix exploitable bugs before they can be exploited. Our work will develop new theoretical and practical models for expressing exploitability (e.g., as predicates over the state space). Thus, our work is of general interest to software development. Task 1.2 looks at finding novel bugs, which is of interest to compilers and formal methods. Our work can also be used for automating exploit generation in general, which is of interest to offensive computing research and national defense. **I have included letters from Symantec and Lockheed-Martin indicating their immediate interest in collaborating in this work if funded.** Our collaborations, if funded, will include shared data sets, access to their in-house experts, and regular meetings to discuss technical details.

Task 2 will secure the patch distribution process and develop new techniques for safe patch application, which will shorten the attack window once a patch has become available. This research will help combat threats that current arise even though patches are available. Our work in secure patch distribution will result in a better understanding of obfuscation, which is of interest in other domains such as digital rights management. Our work in developing filters from patches, if successful, will be of commercial interest to security vendors as it is a new approach for developing filters for their products.

Task 3 will provide an extensible platform for analyzing binary code, which is applicable to the broader computer science audience including model checking and compilers. For example, efficient methods for formally reasoning about code and bad execution states is important to these and other areas. We have collaborations with people in the model checking and theorem proving fields to help transfer ideas between communities. For example, in APEG we worked closely with the author of the decision procedure we used, which led to us creating methods that sped up the decision procedure by up to 200% in our experiments [28].

7 Related Work By Others

Previous work most closely related to our proposed tasks include binary code analysis, symbolic execution, delta debugging, and filter generation.

Binary Analysis. Other researchers have also recognized that source code analysis is insufficient (e.g., [8, 39]). While there are many tools for analyzing binary code (e.g., [57, 123, 131]), most do not represent the code in an explicit and unambiguous IL like BAP, e.g., available architectures we are aware of were not suitable for determining when the jump would be taken in the 3 line program in Figure 3. The most closely related binary analysis platform in the spirit of BAP is CodeSurfer/x86 [8–10]. The published work on CodeSurfer/x86 has focused primarily on alias analysis for assembly [8–10]. Alias analysis is extremely important to scaling any kind of static analysis, and we have successfully implemented their analysis in BAP. Unfortunately, CodeSurfer/x86 is not readily available, thus a more detailed comparison is not possible.

Another related project is Microsoft's next-generation compiler infrastructure, called Phoenix. Phoenix has the ability to read in Microsoft-compiled PE files and then reconstruct a low-level IR when debugging information is available. Phoenix does not aim for faithful binary analysis, e.g., it cannot faithfully analyze the 3 line assembly program in Figure 3 [106]. Phoenix would be an interesting platform to try to integrate a top-down source analysis with a bottom-up binary analysis.

Decompilers (e.g., [39, 41, 61]) raise low-level code up to high-level code, usually employing heuristics to recover information lost during compilation. For example, one could then run source code analysis on the decompiled code [39]. Although decompilers are about recovering higher-level abstractions, which is different than our goal, the general techniques are of interest. For example, Mycroft proposed a type inference algorithm for decompilers, which may be useful strictly at the binary level [111].

Binary instrumentation engines (e.g., [1, 98, 104, 112, 115, 125, 146]) typically do some analysis of binary code in order to find instrumentation points. These tools usually take a purely dynamic approach, while we take a primarily static approach. For example, such tools do not typically perform extensive analysis on how exactly processor flags may be updated. It would be interesting to couple static analysis using BAP with such instrumentation engines.

Translation Validation and Typed Assembly Language. Another complementary approach to fixing the software life-cycle is to first formally verify the high-level source code, then check that compilation preserves all desired properties. Typed assembly language (e.g., [49, 108]) helps verify the compiler by keeping types down to the assembly level. Our work goes up from binary code to an intermediate representation. It would be interesting to see whether we could recover such types, as well as how much such information increased the accuracy of analysis.

Translation validation targets the problem of verifying that an input S to a compiler is semantically equivalent to the output T of the compiler (e.g., [113, 129, 168]). This is a complementary approach when source code is available. Although translation validation has looked primarily at verifying that a compiler optimization is implemented correctly, it would be interesting to adapt it to check if security properties hold before and after an optimization.

Symbolic Execution. Our basic approach generates a formula that is satisfied by all inputs taking a particular path, and then solves the formula to create a specific input that executes the path. This core idea was introduced in the 1970's [77, 90], and since then has appeared in a myriad of applications including automated test case generation (e.g., [13, 15, 34–36, 73–75, 77, 87, 90, 140]), automated signature generation (e.g., [45, 46]), verification (e.g., [65]), automated reverse engineering (e.g., [33, 48]), and fuzzing for security vulnerabilities [72, 75]. Recent work in automated test case generation can produce test cases for a large percentage of program statements (e.g., [34]). This goal is related to ours since determining reachability to possible exploitable statements is important. However, we also want to execute the statements under the *specific conditions necessary to exploit the program*.

There is a wide body of work that confirms the need to reduce false positives in bug reports, e.g., [50, 93, 134]. Work that combines unsound bug-finding with symbolic execution is probably the closest in spirit to our work. This work has primarily been carried out in Java, e.g., Check 'n' Crash [50]. Engler *et al.* propose a statistical rank heuristic, called the z-rank, of whether a bug is likely real or a false positive in Metal/MC [62, 93].

Fuzz-testing can find new bugs even when given only the binary code. For example, fuzz testing has found many integer overflows (e.g., [75, 161]), and found bugs in web-browsers (e.g., [2]). Part of our goal is similar, and we would like to extend these techniques to generate specific exploits.

Delta Debugging. Behavioral difference testing [124], delta debugging [166], and similar work (e.g., [42, 163]) is a developer-oriented technique for pinpointing why a test case failed after changing code. Our work on testing pre-

and post- patch program may benefit from such techniques for explaining why a patch breaks a system. However, our main goal is to show a patch is safe to apply.

Filter Generation. There is considerable work exploring the problem of automatic filter generation for intrusion prevention systems (e.g., [45–47, 51, 89, 92, 102, 103, 118–121, 127, 145, 156, 159, 160, 164, 165]). Sidiroglou *et al.* have proposed that in some cases patches can be generated instead of filters [142–144]. Automatically generated defenses such as these are not intended to replace installing a patch created by the software developer, but are significant and complementary security mechanisms.

8 Integrated Education Plan

My interest in teaching is motivated by the joy of helping students become critical thinkers when using computer science in work and research. My own interest in teaching and research was strongly influenced by courses which required students to constantly evaluate "Why is this secure?". Critically answering such questions about security requires a combination of theory and experimentation.

Reusable Software Security Curriculum. I am setting up a repository for complete software security courses and making it available online at `http://security.ece.cmu.edu`. [2] The curriculum teaches the theory of secure software, along with hands-on labs to reinforce the fundamental concepts.

Lecture Notes on Software Security. Unfortunately, there is no single suitable textbook that explains all the concepts necessary to understand modern software security. For example, someone entering the field of software security should understand symbolic execution, taint analysis, and type safety. Current courses at Carnegie Mellon, and many other institutions I am familiar with, rely on students reading papers in order to glean the fundamental techniques. Reading research papers is important, especially to convey the current state-of-the-art ideas, but is not optimal for teaching the underlying concepts. I am developing a series of lectures notes that teaches many of the important details in software security in a systematic fashion. For example, the notes introduce a small representative language. The language provides a consistent and logical vocabulary throughout the course. We then add elements to the language in order to explain more advanced concept. For example, we show how forward symbolic execution and taint analysis can be built by augmenting an interpreter for the language. One of the goals of our small language is to detail secure coding techniques explicitly in a language-neutral ways.

Hands-on Labs. In my experience, hands-on labs are an effective tool for solidifying theoretical concepts, and helping students develop an understanding for how to combine individual ideas in order to solve real world problems. The challenge when creating labs is to make them interesting, possible within the time allotted, have them directly reinforce concepts from lecture, unambiguous, and be set up so students with variable levels of background knowledge can perform them. I am developing new hands-on labs that integrate with the above lecture notes. The labs reinforce the fundamental concepts and simultaneously show the students how the techniques can be applied to different problem settings. For example, students read about taint analysis for binary code, then show how it could be applied to find SQL injection attacks on the web. We have also designed a software security challenge, which consists of 8 different vulnerable VM's and a centralized automatic grading server. The challenges require students to apply secure coding concepts to languages they have probably never seen before, e.g., we have an input validation vulnerability in the procmail mail filtering language.

I also plan on developing modifications on our next-generation binary analysis platform (Section 5) to help students more easily understand binary analysis, reverse engineering, and exploit authoring. Both the private and public sectors are looking for students with these skills. However, these skills have traditionally been difficult to teach in large part due to the complexity of understanding binary code. Our curriculum uses our binary analysis techniques to reduce this complexity to a simpler set of semantics so that students can focus on core ideas.

Software Security Community Web-Site. Software security is becoming an integral course at many universities. In an effort to create better courses, I have created a website for instructors and students to share information. We will develop modules that incorporate hands-on assignments, and distribute them freely in order to help satisfy demand for software security courses. Instructors can log on and add material, view solutions to previous assignments, and download lecture notes and labs contributed to the site. Students can see previous courses, assignments, and lecture

[2] Although the site is not yet public, it is up and you can log on with account 'instructor' and password 'rotcurtsni'.

notes that have been made available. We plan on adding bulletin-board features for both instructors (e.g., to describe possible lab pitfalls) and students. We have already received a commitment from two professors at UC Santa Barbara in this effort, as noted in the letters of collaboration. We have talked to other universities, such as UC Berkeley, who are also interested in participating. We are also using the website to facilitate our outreach to other universities (described below).

Courses. We are providing the lecture notes and labs described above, along with sample syllabi and course schedules, via the above website. There is currently one set of materials available from the course I developed last semester, and one will be developed in the next semester. Last semester I developed a new course on vulnerability, defense, and malware analysis. For each topic, the course first starts with the best-in-industry tools, and then progresses through the recent research. Each student must complete a course project, which is intended to be a complete rough draft of a publishable idea. The second course focuses on software security in general, and is a core course in the Carnegie Mellon professional Master's program. I am currently bringing it up-to-date by adding new labs focusing on static analysis of source code and binary code, security protection offered by the latest approaches (e.g., stack randomization), and the limitations of those protection schemes (e.g., return-oriented programming [31]).

Development of New Software Security Lab. I believe research informs education. I am developing a software security laboratory which is geared towards educating undergraduate and master's students. The lab will be designed to allow students to safely explore attacks and defenses. The equipment for this lab currently consists of about 15 machines. I have already used this lab as part of the malware analysis course described above, and will continue to use it in my courses, as well as make it available to other instructors at Carnegie Mellon. I am mentoring the Carnegie Mellon offensive computing team, which consists of undergraduates and graduates who enjoy participating in various red-team security contests such as the UC Santa Barbara/USENIX "capture the flag" contest.

Community Outreach. I participate in a continuing Carnegie Mellon information security education outreach program offered to historically Black and Hispanic-serving institutions [153]. This is a two-week long seminar given in the summer each year to faculty at such institutions. The seminar provides tools and teaching techniques for computer security. Partnering institutions this year include Bowie State University, Hampton University, Norfolk State University, and the University of District Columbia.

We also engage in outreach and collaborations with industry. The PI is an active member of Carnegie Mellon's CyLab, a campus-wide initiative on security. We routinely meet with industry partners, and the PI has arranged for collaboration with Symantec and Lockheed-Martin if funded, as described in the included letters of collaboration.

9 Project Timeline

Our timeline tackles focuses on Tasks T1.1 and T2.1 in years 1-3, with a transition to Tasks T1.2 and T2.2 in years 3-5. The Tx.2 tasks also act as a contingency in case we prove negative results on the Tx.1 tasks. We expect to get initial results generating exploits within the first year of Task T1.1, and use later years to investigate more advanced concepts. We expect in Task T2.1 to develop an initial better patch distribution architecture using the proposed multi-stage approach within the first two years. Task 3 is ongoing research performed throughout the 5 year plan, with a up-front focus to bootstrap the other tasks. Our overall timeline in graduate student months is:

	T1.1	T1.2	T2.1	T2.2	T3
Year 1	4		4		4
Year 2	5		4	2	1
Year 3	2	4		4	2
Year 4	1	6		4	1
Year 5	1	4		6	1

10 Prior NSF Support

The PI has not received prior NSF support.

References

[1] On the run - building dynamic modifiers for optimization, detection, and security. Original DynamoRIO announcement via PLDI tutorial, June 2002.

[2] Month of browser bugs website. http://browserfun.blogspot.com, 2006.

[3] The BitBlaze binary analysis project. http://bitblaze.cs.berkeley.edu, 2007.

[4] National Security Agency. *Common Criteria and the Common Evaluation Methodology*, volume Version 3.1, Revision 2. US National Information Assurance Partnership, 2007. Available online at http://www.niap-ccevs.org/cc%2Dscheme/.

[5] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 2nd edition, 2007.

[6] Gerald Aigner, Amer Diwan, David L. Heine, Monica Lam, David L. Moore, Brian R. Murphy, and Constantine Sapuntzakis. An overview of the SUIF2 compiler infrastructure. Technical report, Stanford Computer Systems Laboratory, 2000.

[7] Computing Research Assocation. Grand research challenges in information security and assurance. http://www.cra.org/reports/trustworthy.computing.pdf, Nov 2003.

[8] G Balakrishnan and T Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction*, 2004.

[9] Gogul Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, Computer Science Department, University of Wisconsin at Madison, August 2007.

[10] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86 - a platform for analyzing x86 executables. In *Proceedings of the International Conference on Compiler Construction*, April 2005.

[11] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of CRYPTO*, pages 1–18, 2001.

[12] Armin Biere, Alessandro Bimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[13] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2008.

[14] Nikita Borisov, David Brumley, Heleng Wang, John Dunagan, Pallavi Joshi, and Chuanxiong Guo. A generic application-level protocol analyzer and its language. In *Proceedings of the Network and Distributed System Security Symposium*, 2007.

[15] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on java predicates. In *ACM Internation Symposium on Software Testing and Analysis*, pages 123–133, July 2002.

[16] David Brumley. Automatic generation of multi-path vulnerability signatures for canary. Technical report, Symantec Research Labs, 2007. Unpublished.

[17] David Brumley. *Analysis and Defense of Vulnerabilities in Binary Code*. PhD thesis, Carnegie Mellon University School of Computer Science, 2008.

[18] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the USENIX Security Symposium*, August 2003.

[19] David Brumley and Dan Boneh. Remote timing attacks are practical. *Journal of Computer Networks*, 45(5):701–716, August 2005.

[20] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the USENIX Security Symposium*, Boston, MA, August 2007.

[21] David Brumley, Tzi cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. Efficient and accurate detection of integer-based attacks. In *Proceedings of the Network and Distributed System Security Symposium*, 2007.

[22] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Dawn Song. Bitscope: Automatically dissecting malicious binaries. Technical Report CS-07-133, School of Computer Science, Carnegie Mellon University, March 2007.

[23] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In Wenkee Lee, Cliff Wang, and David Dagon, editors, *Botnet Detection*, volume 36 of *Countering the Largest Security Threat Series: Advances in Information Security*. Springer-Verlag, 2008.

[24] David Brumley, Li-Hao Liu, Pongsin Poosank, and Dawn Song. Design space and analysis of worm defense systems. Technical Report CMU-CS-05-156, Carnegie Mellon University, 2005.

[25] David Brumley and James Newsome. Alias analysis for assembly. Technical Report CMU-CS-06-180, Carnegie Mellon University School of Computer Science, 2006.

[26] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 2–16, 2006.

[27] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Theory and techniques for automatic generation of vulnerability-based signatures. *IEEE Transactions on Dependable and Secure Computing*, 5(4):224–241, October 2008.

[28] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2008.

[29] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the USENIX Security Symposium*, 2004.

[30] David Brumley, Hao Wang, Somesh Jha, and Dawn Song. Creating vulnerability signatures using weakest pre-conditions. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2007.

[31] Erik Buchanan, Ryan Roemer, Hovav Shcham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 27–38, 2008.

[32] Juan Caballero and Dawn Song. Rosetta: Extracting protocol semantics using binary analysis with applications to protocol replay and NAT rewriting. Technical Report CMU-CyLab-07-014, Carngie Mellon University CyLab, October 2007.

[33] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, October 2007.

[34] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2008.

[35] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the International SPIN Workshop on Model Checking of Software*, 2005.

[36] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the ACM Conference on Computer and Communications Security*, October 2006.

[37] Internet Storm Center. Survival time. http://isc.sans.org/survivaltime.html, Last updated 29 Feb 2007. URL Checked 18 Jun 2009.

[38] CERT/CC. Advisory ca-2001-19: "code red" worm exploiting buffer overflow in IIS indexing service dll. http://www.cert.org/advisories/CA-2001-19.html, 2001.

[39] Bor-Yun Evan Chang, Matthew Harren, and George Necula. Analysis of low-level code using cooperating decompilers. In *Proceedings of the Static Analysis Symposium*, 2006.

[40] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*, August 2004.

[41] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, July 1994.

[42] Holger Cleve and Andreas Zeller. Locating casues of program failures. In *Proceedings of the ACM Conference on Software Engineering*, 2005.

[43] Intel Corporation. Tamper resistant methods and apparatus, 2001. US Patent 6,205,550.

[44] Mitre Corporation. CWE-14: Compiler removal of code to clear buffers. http://cwe.mitre.org/data/definitions/14.html. URL checked 1/26/2009.

[45] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of the ACM Symposium on Operating System Principles*, October 2007.

[46] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the ACM Symposium on Operating System Principles*, 2005.

[47] Jedidiah Crandall, Zhendong Su, S. Felix Wu, and Frederic Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2005.

[48] Jedidiah R. Crandall, Gary Wassermann, Daniela de Oliveira, Zhendong Su, S. Felix Wu, and Fredric Chong. Temporal search: Detecting hidden malwre timebombs with virtual machines. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[49] Karl Crary. Toward a foundational typed assembly language. In *Symposium on Principles of Programming Languages*, 2003.

[50] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: Combining static checking and testing. In *Proceedings of the ACM Conference on Software Engineering*, 2005.

[51] Weidong Cui, Marcus Peinado, Helen J. Wang, and Michael Locasto. Shi eldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[52] CMU CyLab. http://www.cylab.cmu.edu.

[53] Apache 2.0 Bugzilla Database. http://issues.apache.org/bugzilla. Open, non-trivial and non-enhancement bugs. Checked 6/26/09.

[54] Firefox 3.0 Bugzilla Database. http://bugzilla.mozilla.org. Open, non-trivial and non-enhancement bugs. Checked 6/26/09.

[55] Linux 2.6 Bugzilla Database. http://bugzilla.kernel.org. Open, non-trivial and non-enhancement bugs. Checked 6/26/09.

[56] Ubuntu Bugzilla Database. http://launchpad.ubuntu.com. Open, non-trivial and non-enhancement bugs. Checked 6/26/09.

[57] DataRescue. IDA Pro. http://www.datarescue.com. URL checked 7/31/2008.

[58] Google Chrome Developers. Software updates: Courgette. http://dev.chromium.org/developers/design-documents/software-updates-courgette, 2009. Page checked 7/19/09.

[59] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[60] eEyE Security. eEye binary diffing suite (EBDS). http://research.eeye.com/html/tools/RT20060801-1.html. Version 1.0.5.

[61] Michael James Van Emmerik. *Single Static Assignment for Decompilation*. PhD thesis, The University of Queensland School of Information Technology and Electrical Engineering, May 2007.

[62] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2000.

[63] Dawson Engler and Madanlal Musuvathi. Static analysis versus software model checking for finding bugs. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, 2004.

[64] Dennis Fischer. New linux flag enables null pointer exploits. http://threatpost.com/blogs/researcher-uses-new-linux-kernel-flaw-bypass-selinux-other-protections, 2009. Page checked 7/18/09.

[65] C. Flanagan and J.B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the Symposium on Principles of Programming Languages*, 2001.

[66] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2002.

[67] Jeff Foster, Rob Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *Transactions on Programming Languages*, 28(6):1035–1087, 2006.

[68] Jeffrey Foster, Manuel Fahndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1999.

[69] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the ACM Conference on Software Engineering*, 2008.

[70] Debian Gao, Michael Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *Proceedings of the International Conference on Information and Communications Security*, pages 238–255, October 2008.

[71] Christos Gkantsidis, Thomas Karagiannis, Pablo Rodriguez, and Milan Vojnovic. Planet scale software updates. In *Proceedings of the ACM Special Interest Group on Data Communication*, 2006.

[72] Patric Godefroid, Adam Kiezun, and Michael Levin. Grammar-based whitebox fuzzing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 206–215, 2008.

[73] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the Symposium on Principles of Programming Languages*, 2007.

[74] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2005.

[75] Patrice Godefroid, Michael Levin, and David Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, February 2008.

[76] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the USENIX Security Symposium*, February 2008.

[77] K. Hanford. Automatic generation of test cases. *IBM SYstems Journal*, 9(4), 1970.

[78] Thomas Hart, Kelvin Ku, David Lie, Marsha Chechik, and Arie Gurfinkel. Ptyasm: Software model checking with proof templates. In *Proceedings of the IEEE/ACM Automated Software Engineering Conference*, 2008.

[79] Brian Hatch. Stunne: RSA timing attacks/key discovery. http://marc.info/?l=stunnel-users&m=104827610907579&w=2, 2003.

[80] John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introduction to automata theory, langauges, and computation*. Addison-Wesley, 2001.

[81] Michael Howard. Some bad news and some good news. http://msdn2.microsoft.com/en-us/library/ms972826.aspx, 2002.

[82] Daniel Jackson and Eugene J. Rollins. Chopping: A generalization of slicing. Technical Report CS-94-169, Carnegie Mellon University School of Computer Science, 1994.

[83] Matthias Jacob, Dan Boneh, and Edward Felten. Attacking an obfuscated cipher by injecting faults. In *Digital Rights Management*, pages 16–31. Springer LCNS, 2003.

[84] Havard Johansen, Dag Johansen, and Robbert van Renesse. Firepatch: Secure and time-critical dissemination of software patches. In *IFIP International Information Security Conference*, May 2007.

[85] Robert Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the USENIX Security Symposium*, 2004.

[86] Patrick R. Gallagher Jr. A guide to understanding object reuse in trusted systems. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.5809, 1992. Page checked 7/18/09.

[87] Sarfraz Khurshid and Darko Marinov. TestEra: a novel framework for automated testing of java programs. In *Proceedings of the Conference on Automated Software Engineering*, 2001.

[88] Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 243–256, 2007.

[89] Hyang-Ah Kim and Brad Karp. Autograph: toward automated, distributed worm signature detection. In *Proceedings of the USENIX Security Symposium*, 2004.

[90] James King. Symbolic execution and program testing. *Communications of the ACM*, 19:386–394, 1976.

[91] Paul Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In *Proceedings of CRYPTO*, pages 104–113, 1996.

[92] Christian Kreibich and Jon Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proceedings of the ACM Workshop on Hot Topics in Networks*, 2003.

[93] Ted Kremenek and Dawson Engler. Z-ranking: Using statical analysis to counter the impact of static analysis. In *SAS*, pages 295–315, 2003.

[94] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the USENIX Security Symposium*, 2005.

[95] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the USENIX Security Symposium*, 2004.

[96] Monica Lam, Constantine Sapuntzakis, Ramesh Chandra, David Brumley, Nickolai Zeldovich, Mendel Rosenblum, and James Chow. A cache-based system management architecture with virtual appliances, network repositories, and virtual appliance transceivers. U.S. Patent, May 2008. Patent 7373451.

[97] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):327–337, December 1992.

[98] James Larus and Eric Schnarr. EEL: machine-independent executable editing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 291–300, 1995.

[99] Chris Lattner. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the Symposium on Code Generation and Optimization*, 2004.

[100] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[101] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: a tool for finding copy-paste related bugs in operating system code. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2004.

[102] Zhichun Li, Manan Shanghi, Brian Chavez, Yan Chen, and Ming-Yang Kao. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resiliance. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[103] Zhenkai Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2005.

[104] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2005.

[105] Microsoft. Dotfuscator community edition 4.0. http://msdn.microsoft.com/en-us/library/ms227240.aspx. URL checked 7/10/2009.

[106] Microsoft. Phoenix project architect posting. http://forums.msdn.microsoft.com/en-US/phoenix/thread/90f5212c-f05a-4aea-9a8f-a5840a6d101d, July 2008. URL checked 7/31/2008.

[107] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. In *Proceedings of the IEEE Symposium on Security and Privacy*, volume 1, 2003.

[108] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Language Systems*, 21(3):527–568, 1999.

[109] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the USENIX Security Symposium*, 2007.

[110] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[111] Alan Mycroft. Type-based decompilation. In *European Symposium on Programming*, March 1999.

[112] Susanta Nanda, Wei Li, Lap chung Lam, and Tzi cker Chiueh. BIRD: Binary interpretation using runtime disassembly. In *Proceedings of the IEEE/ACM Conference on Code Generation and Optimization*, March 2006.

[113] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2000.

[114] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the International Conference on Compiler Construction*, 2002.

[115] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation or Building Tools is Easy*. PhD thesis, Trinity College, University of Cambridge, 2004.

[116] James Newsome, David Brumley, Jason Franklin, and Dawn Song. Replayer: Automatic protocol replay by binary analysis. In Rebecca Write, Sabrina De Capitani di Vimercati, and Vitaly Shmatikov, editors, *Proceedings of the ACM Conference on Computer and Communications Security*, pages 311–321, 2006.

[117] James Newsome, David Brumley, and Dawn Song. Sting: An end-to-end self-healing system for defending against zero-day worm attacks. Technical Report CMU-CS-05-191, Carnegie Mellon University School of Computer Science, 2006.

[118] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[119] James Newsome, Brad Karp, and Dawn Song. Paragraph: Thwarting signature learning by training maliciously. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, September 2006.

[120] James Newsome, David Brumleyand Dawn Song, Jad Chamcham, and Xeno Kovah. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proc. of the* 13^{th} *Annual Network and Distributed System Security Symposium (NDSS)*, 2006.

[121] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, February 2005.

[122] OpenSSL Security Advisory. Timing-based attacks on RSA keys. http://www.openssl.org/news/secadv_20030317.txt, 2003. URL checked on 6/16/09.

[123] Alex Opterix. Binstat. http://rpisec.net/projects/show/rcosbinstat. Page checked 7/19/09.

[124] Alessandro Orso and Tao Xie. BERT: Behavioral regression testing. In *Proceedings of the Workshop on Dynamic Analysis*, 2008.

[125] Paradyn/Dyninst. Dyninst: An application program interface for runtime code generation. http://www.dyninst.org. URL checked 9/28/2008.

[126] Colin Percival. *Matching and Mismatches and Assorted Applications*. PhD thesis, Wadham College, University of Oxford, 2006. bsdiff/bspatch tool.

[127] Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Fogla, and Monirul Sharif. Misleading worm signature generators using deliberate noise injection. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2006.

[128] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide*, 1.2 edition, September 1998. Available as Technical Report CMU-CS-98-173, Carnegie Mellon University.

[129] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166, 1998.

[130] The Snort Project. Snort, the open-source network intrusion detection system. http://www.snort.org/.

[131] Daniel Quinlan and Thomas Panas. Source code and binary analysis of software defects. In *Proceedings of the Workshop on Cyber Security and Information Intelligence Research*, 2009. Article 40, Track 4.

[132] Project Manager Robert Martin. CWE/SANS top 25 most dangerous programming errors. http://www.sans.org/top25errors/, 2009. URL Checked 7/9/09.

[133] Jim Roskind. Attacks against the netscape browser plus security response philosophy and methods. Private communication and seminar talk.

[134] Nick Rutar, Christian Almazan, and Jeffrey Foster. A comparison of bug finding tools for java. In *Proceedings of the IEEE Symposium on Softare Reliability Engineering*, pages 245–256, 2004.

[135] Sabre Security. Bindiff. http://www.sabre-security.com/products/bindiff.html.

[136] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M.S. Lam, and M. Rosenblum. Virtual applicances for deploying and maintaining software. In *Proceedings of the USENIX Large Installation System Administration Conference*, 2003.

[137] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution of binary programs. In *International Symposium on Software Testing and Analysis*, 2009.

[138] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[139] Secure Science Corporation. Analysis of the WebViewFolderIcon ActiveX integer overflow (setSlice). http://www.mnin.org, 2006.

[140] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for c. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering*, 2005.

[141] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[142] Stelios Sidiroglou and Angelos D. Keromytis. A network worm vaccine architecture. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, Workshop on Enterprise Security*, pages 220–225, June 2003.

[143] Stelios Sidiroglou and Angelos D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, 2005.

[144] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a reactive immune system for software services. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[145] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. The EarlyBird system for real-time detection of unknown worms. Technical Report CS2003-0761, University of California, San Diego, August 2003.

[146] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[147] G. Edward Suh, Jaewook Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[148] Symantec. Blaster worm. http://www.symantec.com/security_response/writeup.jsp?docid=2003-081113-0229-99, 2003.

[149] Microsoft TechNet. Conficker worm: Help protect windows from conficker. http://technet.microsoft.com/en-us/security/dd452420.aspx, Feb 2009.

[150] Oreans Technology. Themida. http://www.oreans.com/. URL checked 7/10/2009.

[151] Julien Tinnes and Tavis Ormandy. Bypassing linux' NULL pointer dereference exploit prevention. http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html, 2009. Page checked 7/18/09.

[152] Joseph Tucek, James Newsome, Shan Lu, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou, and Dawn Song. Sweeper: A lightweight end-to-end system for defending against fast worms. In *Proceedings of the EuroSys Conference*, 2007.

[153] Carnegie Mellon University. Information assurance capacity building program. http://www.cit.cmu.edu/about_cit/events/2009/07_13_iacbp.html. Page checked 7/20/2009.

[154] US-CERT. Crypto++ information for vu#997481. http://www.kb.cert.org/vuls/id/AAMN-5K52DH, 2003.

[155] Michael Venable, Mohamed R. Chouchane, Md Enamul Karim, and Arun Lakhotia. *Analyzing Memory Accesses in Obfuscated x86 Executables*, volume 3548 of *Intrusion and Malware Detection and Vulnerability Assessment*. Lecture Notes in Computer Science, 2005.

[156] Shobha Venkataraman, Avrim Blum, and Dawn Song. Limits of learning-based signature generation with adversaries. In *Proceedings of the Network and Distributed System Security Symposium*, February 2008.

[157] Milan Vojnovic and Ayalvadi Ganesh. On the race of worms, alerts and patches. *IEEE/ACM Transactions on Networking*, 16(5):1066–1079, 2008.

[158] David Wagner, Jeffrey Foster, Eric Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, 2000.

[159] Helen J Wang, Chuanxiong Guo, Daniel Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the ACM Special Interest Group on Data Communication*, August 2004.

[160] Ke Wang, Janak Parekh, and Salvatore J. Stolfo. Anagram: A content anonaly detector resistant to mimicry attack. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 226–248, September 2006.

[161] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium*, 2008.

[162] Mark Weiser. Program slicing. In *Proceedings of the Conference on Software Engineering*, pages 142–151, 1981.

[163] Joel Winstead and David Evans. Towards differential program analysis. In *Proceedings of the Workshop on Dynamic Analysis*, 2003.

[164] Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Chris Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2005.

[165] Vinod Yegneswaran, Jonathon T. Giffin, Paul Barford, and Somesh Jha. An architecture for generating semantics-aware signatures. In *Proceedings of the USENIX Security Symposium*, 2005.

[166] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2), 2002.

[167] Jiang Zheng. *Buffer Overflow Vulnerability Diagnosis for Commodity Software by Binary Analysis*. PhD thesis, University of Pittsburgh, 2008.

[168] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, March 2003.