The Application of Top-Down Abstraction Learning Using Prediction as a Supervisory Signal to Cyber Security

Jonathan Mugan and Aram Khalili

21CT, Inc., Austin, Texas, USA
www.21ct.com

ABSTRACT

Current computer systems are dumb automatons, and their blind execution of instructions makes them open to attack. Their inability to reason means that they don't consider the larger, constantly changing context outside their immediate inputs. Their nearsightedness is particularly dangerous because, in our complex systems, it is difficult to prevent all exploitable situations. Additionally, the lack of autonomous oversight of our systems means they are unable to fight through attacks. Keeping adversaries completely out of systems may be an unreasonable expectation, and our systems need to adapt to attacks and other disruptions to achieve their objectives. What is needed is an autonomous controller within the computer system that can sense the state of the system and reason about that state.

In this paper, we present Self-Awareness Through Predictive Abstraction Modeling (SATPAM). SATPAM uses prediction to learn abstractions that allow it to recognize the right events at the right level of detail. These abstractions allow SATPAM to break the world into small, relatively independent, pieces that allow employment of existing reasoning methods. SATPAM goes beyond classification-based machine learning and statistical anomaly detection to be able to reason about the system, and SATPAM's knowledge representation and reasoning is more like that of a human. For example, humans intuitively know that the color of a car is not relevant to any mechanical problem, and SATPAM provides a plausible method whereby a machine can acquire such reasoning patterns. In this paper, we present the initial experimental results using SATPAM.

Keywords: machine learning, automation, automation assurance, autonomic computing, verification and validation, cyber resilience, causal models

1. INTRODUCTION

Our computer networks form a complex system. At each point in time, our systems are in a particular state and they step to the next state either by receiving input or by carrying out previously specified instructions. Sequences of steps are paths, with each path being a possible future. There are far too many paths to consider them all, and hackers look for vulnerable paths to compromise systems. When vulnerable paths are found and exploited by adversaries, defenders respond to changing the system to block those paths, leading to an arms race.

Staying ahead of the hackers in the search for vulnerable paths is difficult, and even biology is not immune. Alcon butterfly caterpillars trick ants into protecting and providing for their young. They hack the ants by forging chemical signatures used by ants to identify baby ants.[1] Even a system as sophisticated as the mammalian brain can be hacked. There is a parasite called Toxoplasma that is primarily spread through cat feces. Normally frightened by cat urine, rats infected with Toxoplasma actually seek out cat urine. Toxoplasma hijacks the sexual circuits of the rats brain so that they are sexually attracted to cat urine, making it more likely for these rats to be eaten by cats, and therefore, more likely to spread the parasite.[2]

To find these vulnerable paths before the hackers do, we need the ability to automatically search for them. We also need to give our computers the autonomy to defend themselves when new vulnerable paths are found. There has been some excellent work on automatically generating exploits and malware. MAYHEM[3] automatically generates exploits in binary code. MAYHEM looks for sections of binary code that are dependent upon user

Send correspondence to Jonathan Mugan at jmugan@21ct.com.

input, potentially from an attacker. When it finds such a section of binary code, it uses symbolic execution to try to generate an exploit. Frankenstein[4] stitches together little pieces of trusted binary code to re-implement malware. Since the code of the malware matches the structure of the computer system from which it came, the "immune system" of the computer system cannot detect it as malware. GenProg[5] uses genetic programming to automatically fix software bugs. If you have a set of test cases, GenProg can search for a program change (patch) that enables the program to pass all test cases. ClearView[6] is closest to the work presented in our paper. ClearView automatically finds errors and applies patches by learning invariants. It learns invariants by watching running programs, and it then notes when those invariants are violated during failed executions.

The work presented in this paper is less mature in the cyber domain than much of the previous work in the area, but it represents a very ambitious approach. Instead of being grounded in cyber, our work stems from work in developmental robotics. Our algorithm, Self-Awareness Through Predictive Abstraction Modeling (SATPAM), monitors the system as it runs. Through experience, it learns abstractions that help it more effectively see vulnerable paths. SATPAM then has the the causal understanding and the reasoning ability to determine when the system is in a vulnerable state.

2. SATPAM

SATPAM is built on top of the Qualitative Learner of Action and Perception (QLAP).[7] QLAP is a developmental learning algorithm that begins with very little knowledge of the environment and autonomously learns through exploration. QLAP simultaneously learns abstractions of the environment and models based on those abstractions, with new knowledge being built on top of old knowledge. The models that QLAP uses are dynamic Bayesian networks (DBNs).[8] DBNs are a type of graphical model that is well suited for representing knowledge that is causal and temporal in nature. In QLAP, each DBN predicts an event on some variable where an event is a change in the value of that variable. QLAP also learns context variables for DBNs which specify the situations in which they are reliable. The *reliability* of a DBN is how often the predicted event actually occurs in the current context. If there is a context for which the reliability of a DBN is greater than 0.75, we call that DBN *reliable*. Reliable DBNs are used for planning.

In our previous work, Cy-QLAP[9] extended QLAP by porting the learning agent to the cyber security domain. SATPAM extends QLAP and Cy-QLAP by expanding the type of abstractions that can be learned. SATPAM defines domain-specific abstraction hierarchies that represent abstractions on a continuum from high-level to low-level. An example of a high-level abstraction would be representing the event that there was a change in a configuration file. An example of a low-level abstraction would be representing the more specific event that field x in configuration file y was changed to value v. In between these two extremes, one could represent only that *some* value was changed in configuration file y. The lower the agent goes on the abstraction hierarchy, the more detail it can use for learning models, but going lower on the abstraction hierarchy has the disadvantage of increasing the size of the search space. The details of the SATPAM abstraction learning architecture are provided in our workshop paper.[10] This paper presents preliminary experimental results, as discussed below.

3. EXPERIMENTAL DOMAIN: CODE INJECTION ARMS RACE

We evaluate SATPAM in the domain of code injection. Code injection is a way of changing a system through an interaction point (attack vector). An attack vector is an entry point into a computer system. Various attack vectors include:

Accepting user input: This can be from a web form or a web service. User input leaves the technology open to buffer overflow attacks. Malicious code is injected with a buffer overflow, often in the form of string input.

Visiting a website: A user can be tricked into going to a bad website that causes attacker JavaScript to be executed by the user's browser. When a user visits a website, the browser downloads everything on the page, including images, which can be as small as one pixel.

Inadvertently downloading a malicious program: A user can be tricked into downloading and executing a malicious program, usually through phishing attacks.

Inserting a thumb drive: A user inserts a thumb drive with malicious content.

Over the last thirty years, we have been in the middle of a code injection arms race.* A buffer overflow occurs when an input is provided to a program that is larger than the buffer allocated to hold that input. Program variables are stored on a stack, and the result of a buffer overflow is that the stack is overwritten with attacker-provided input. This input can overwrite the return address of the function so that the return address points to code supplied within the input that overflowed the buffer. When the subroutine reaches the part where it usually returns to its calling subroutine, the program instead returns to the code given in the input that overflowed the buffer.

In the remainder of this section, we provide an informal overview of the progression of the code injection arms race.

3.1 Defense: Data Execution Prevention (DEP)

To defend against a buffer overflow attack, defenders started marking parts of the stack as either writable or executable, but not both. This is called DEP or XxorW, and it prevents the computer from executing code that has been injected (written) by an attacker. In Windows, DEP became available in Windows XP (service pack 2), and DEP made it into the Linux kernel 2.6.8 in 2004.

3.2 Attack: Return to `libc`

To overcome the defense of XxorW (DEP), attackers changed their buffer overflow tactics. Instead of overwriting the return address to point to code they injected, attackers overwrote the return address to be the `exec()` function inside of a system library such as `libc`.

3.3 Defense: Address Space Layout Randomization (ASLR)

To defend against the "return to `libc`" attack, defenders began randomly scrambling the addresses of the system functions, meaning that attackers could not know where `exec()` in `libc` would be ahead of time. This defense is particularly interesting because what makes computers vulnerable is their consistency. The internals are complicated, but attackers only have to figure them out once. ASLR adds artificial diversity. ASLR was implemented in Windows Vista. In Linux, a form of ASLR was the default in kernel version 2.6.12, released in 2005, and it was also made available via the PaX patchset.

3.4 Attack: Return Oriented Programming

"Return to `libc`" was a precursor to return oriented programming because an attacker can string together multiple calls to portions of system code. Return oriented programming searches system binaries for little pieces of code that end in a return instruction, called *gadgets*.[4]

4. EXPERIMENTAL PROCEDURE

Our experiment is based on the buffer overflow vulnerability of non-memory safe programming languages, in our case the C programming language. During a buffer overflow, if the return address is overwritten with a random value, the program is likely to crash when it tries to return from that subroutine. SATPAM has learned that the reliable predictor of a crash is when the return address changes but the frame address does not change; we refer to this state as *unstable*. Using learning and reasoning, the system monitor tracks the current state of the system and is able to predict when this unstable state will come about by reasoning using backward chaining. If SATPAM predicts the system will go into a bad state, where a buffer overflow can occur and the system can be compromised, it stops execution. In this section, we first talk about how the learning is done, and then we discuss how SATPAM defends the system.

*For background, see a course such as Kevin Hamlen's Language-based Security `http://www.utdallas.edu/~kxh060100/cs6V81fa12.html`, and in particular see the guest lecture material of Wartell et al., `http://www.utdallas.edu/~zxl111930/file/CCS12.pptx`.

4.1 Training the System

We executed the following steps to learn the system dynamics in the experiment.

1. Generate 100 random programs as described in Section 4.3.
2. Execute each program in the gdb debugger and extract features for each line as described in Section 4.4.
   - With probability 0.5 execute the program with a buffer overflow, and with probability 0.5 execute the program without a buffer overflow. Execution with a buffer overflow means that the input string to the program is larger than the program can hold.
3. Use this data to learn predictive models in the form of DBNs, as done in QLAP.[7†]

4.2 Defending the System

After, training, we evaluate how well the learned DBNs can defend the system by predicting when a current running program is subject to a buffer overflow. The steps for doing this are as follows:

1. Generate 100 new random programs as described in Section 4.3.
2. Execute each in the debugger twice: once with a buffer overflow, and once without a buffer overflow. In both cases, record features for each line as described in Section 4.4.
3. Attach SATPAM to each program. SATPAM decides whether the program has a buffer overflow condition by searching for a reasoning path from a current state to an unstable state as it monitors the program. SATPAM searches for this path using backchaining.[9] If such a path is found, it determines that the program has a buffer overflow.

4.3 Program Generator

For training and testing data, we created a program generator that generates C programs. Each program has a primary `main()` function, and a dummy function that only assists in setting an entry breakpoint in the debugger (cf. below). The `main()` function declares a number of integer and character array variables statically and performs a number of integer assignments and string copies. An example generated program is shown in Listing 1.

[†]QLAP first adds additional variables to the DBN by hillclimbing on reliability until the reliability of the DBN reaches a threshold of 0.75. After that point, QLAP hillclimbs on the entropy of the conditional probably table of the DBN. In this work, we found that the best results came from only hillclimbing on the entropy of the conditional probability table, and that is the method we used in this paper.

Listing 1: Example Program

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void func1(void)
{
}

int main(int argc, char **argv)
{
  int aa = 0;

  char buffer1[10];
  char buffer2[10];
  char buffer3[10];
  char buffer4[10];
  char buffer5[10];
  char buffer6[10];
  char buffer7[10];
  char buffer8[10];
  char buffer9[10];

  func1();

  aa = 1240842111;
  aa = 711615867;
  strcpy(buffer8, argv[1]);
  strcpy(buffer7, argv[1]);
  strcpy(buffer4, argv[1]);
  aa = 277241113;
  strcpy(buffer4, argv[1]);
  strcpy(buffer9, argv[1]);
  aa = 1199317997;
  aa = 1585555772;

}
```

4.4 Extracting Features from a Running Program

Our gdb/Python instrumentation runs each generated program and collects information on the stack frame, the variable values in the current context, and the current line of source code. The current line of source code is available because we created the executable from source code that we generated and compiled with additional debug information enabled.

In programs created by the generator, the integer variables are of the form `aa`, `bb`, `cc`, etc.; and the string variables are of the form `buffer1`, `buffer2`, `buffer3`, etc. Our gdb/Python framework extracts the value of these variables, and whether they were referenced in the current line of source code. From this, SATPAM preprocessing creates features that are true when a particular variable is changed, e.g. `aa_change, buffer1_change`, etc. Additionally we create a feature, `var_change_without_reference`, that is true when any variable is changed without being referenced in the current line of source code. The integer variables are always declared before the character array, and right after the function begins. In this way, the variable `aa` is always the first variable on

Table 1: Features Extracted from Each Line of Code as It Runs

Feature	Description
current_function_change	True when then name of the current function being executed changed value from last timestep.
saved_ebp_change	True when EBP register (stack base pointer) value changed from last timestep.
saved_ebx_change	True when EBX register (general purpose register) value changed from last timestep.
saved_edi_change	True when EDI register (destination index) value changed from last timestep.
saved_eip_change	True when EIP register (instruction pointer) value changed from last timestep.
saved_esi_change	True when ESI register (source index) value changed from last timestep.
frame_address_change	True when address of current stack frame changed from last timestep.
local_variables_start_address_change	True when start address of local variables in current stack frame changed from last timestep.
previous_frame_address_change	True when address of previous stack frame (parent function) changed from last timestep.
previous_frame_stack_pointer_change	True when value of stack pointer of previous stack frame (parent function) changed from last timestep.
program_change	True when the name of the current program being executed changes from the last timestep.
return_address_change	True when return address changed from last timestep.
stacklevel_change	True when stacklevel changed from last timestep.
xx_change	True when variable xx changed from last timestep. The variables names are aa through ii and buffer1 through buffer9. (Variable ff is not used to keep the name from being part of buffer.).
xx_referenced	True when variable xx is referenced in the current line of code.
vars_change_but_not_referenced	True when any of source program's variables have changed value from last timestep, and not been referenced during timestep.
block_end	True when source code line includes "}".
unstable	True when the program becomes unstable.
crash	True when the program crashes.

the stack after the return address. Table 1 shows the features extracted from each line of code. These are the features (variables) on which the DBNs are learned.[‡]

5. EXPERIMENTAL RESULTS

In this section, provide quantitative results and discuss learned abstractions.

[‡]For these variables, we are only concerned when they become True and so DBNs are only learned on events where they become true. In addition, we do not allow crash to be on antecedent of a contingency because a program stops executing at that point.

5.1 Quantitative Results

During training, SATPAM learned 187 reliable DBNs. For evaluation, there were 100 training programs and 100 testing programs. We ran the evaluation on those programs ten times, resulting in 1000 buffer overflow cases, 1000 cases with no buffer overflow, and 2000 runs total. We specify the following cases:

- true positive (TP): buffer overflow and identified
- false positive (FP): no buffer overflow and identified
- true negative (TN): no buffer overflow and not identified
- fasle negative (FN): buffer overflow and not identified

The results were

- Number of TP: 608
- Number of FP: 65
- Number of TN: 935
- Number of FN: 392

Looking a little more deeply at these results, we see that

- Accuracy (TP + TN)/(TP + FP + TN + FN): (608 + 935) / (608 + 65 + 935 + 392) = 0.772
- Precision (TP / TP + FP): 608 / (608 + 65) = 0.903
- Recall (also known as sensitivity or true positive rate) (TP / TP + FN): 608 / (608 + 392) = 0.903
- Specificity (also known as true negative rate) (TN / TN + FP): 935 / (935 + 65) = 0.935

5.2 Abstraction Learning

SATPAM learns two kinds of abstractions. *Focus-of-attention abstractions* tell the controller which variables it should pay attention to. SATPAM pays attention to the variables that show up in reliable DBNs. The reasoning is that DBNs are used to make predictions, and if there is some variable that cannot be predicted and cannot be used to make predictions, then there is no need for SATPAM to pay attention to it.

SATPAM learned focus-of-attention abstractions in this experiment. For example, SATPAM learned that the system can come into an unstable state if a program variable is referenced in a line of code and executing that line of code results in the return address on the stack changing. An example DBN of this form learned is

`buffer1_referenced --> unstable` when `return_address_change` (reliability 1.0).

Focus of attention abstractions are important because computer systems are too complicated for a learning agent to pay attention to everything, and the SATPAM must therefore learn what is important.

The second kind of abstraction learned by SATPAM is *level-of-detail abstractions*. Level-of-detail abstractions tell the controller the resolution at which it should be paying attention. In the previous example, we saw that it learned that if the specific variable `buffer1` was in a line of code, the controller should pay attention. However, this is overly specific. Fortunately, SATPAM also learns that if the value of any program variable on the stack changes when a line of code is executed, and that program variable does not exist in the line of code, and the return address changes, this can lead to an unstable state:

`var_changed_not_referenced --> unstable` when `return_address_change` (reliability 0.89).

This more general rule allows SATPAM to move up one level of detail so that its learning and knowledge representation are more efficient.

6. CONCLUSION

By monitoring running programs, SATPAM has learned how to identify simple buffer overflow attacks. Of course, defenses for simple buffer overflow attacks already exist, as we saw in Section 3, but what is important here is that SATPAM *learned* this by monitoring running programs. By using learning, SATPAM holds the potential to find dangers before they are widely known. SATPAM could also be used to learn policies. Schneider[11] discusses the importance of security policies, and such policies could be learned with SATPAM. For example, SATPAM learns that if `vars_changed_but_not_referenced` becomes True and `return_address_change` is True, then it is best for the program to halt. Learned policies such as this one could be placed inline in the code by the compiler.

The speed and efficacy of developmental learning could be increased by incorporating known attack patterns. Outside knowledge can come in the form of state transitions from modeled attacks using the Nessus vulnerability scanner.[12] Nessus remotely checks for vulnerabilities using a set of known possible exploits. Nessus[§] can serve a similar function as the open source Metasploit Framework,[¶] which consists of a set of known attacks that can be used to check the security of a system to either protect it or compromise it. Another potential source of external information could come from the National Vulnerability Database.[‖]

It is expensive to attach to a program and monitor it line by line, and the decision of when to attach SATPAM to a program needs to be made intelligently. Future work could focus on increasingly high-level decisions by the controller, such as which program needs to be monitored. We could also expand the kinds of situations where SATPAM could defend a system. For example, SQL injection is another often-exploited vulnerability. We could perform an experiment where SATPAM seeks to learn an abstraction that helps it efficiently learn when inputs are dangerous. In particular, SATPAM could learn which characters are dangerous to have in a SQL statement. SATPAM would likely learn that the category of all non-letter characters is too general, and it would likely focus on dangerous characters such as "`;`" (statement terminator) and "`--`" (comment).

The results presented here are a modest beginning. The programs that SATPAM protects were generated randomly within a narrow range of syntaxes. However, the approach of autonomously searching for causal rules that can be used to model the system is an important piece to solving the puzzle of how to defend such systems. As our computer systems become more complicated, manually searching for vulnerabilities will become increasingly infeasible. Technological superiority will go to those who have the most effective automation.

ACKNOWLEDGMENTS

The author would like to thank the Office of Naval Research (ONR) for their generous support. In particular, the author would like to thank Sukarno Mertoguno, Myong Kang, and Jim Luo for their guidance and support.

REFERENCES

[1] Attenborough, D., "Life in the undergrowth." BBC One (2005).

[2] House, P., Vyas, A., and Sapolsky, R., "Predator cat odors activate sexual arousal pathways in brains of toxoplasma gondii infected rats," *PLoS ONE* **6**(8) (2011).

[3] Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D., "Unleashing mayhem on binary code," in [*Security and Privacy (SP), 2012 IEEE Symposium on*], 380–394, IEEE (2012).

[4] Mohan, V. and Hamlen, K. W., "Frankenstein: Stitching malware from benign binaries," in [*Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*], 77–84 (August 2012).

[5] Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W., "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in [*Software Engineering (ICSE), 2012 34th International Conference on*], 3–13, IEEE (2012).

[6] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., et al., "Automatically patching errors in deployed software," in [*Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*], 87–102, ACM (2009).

[§] `http://www.tenable.com/products/nessus`

[¶] `http://www.metasploit.com/`

[‖] `http://nvd.nist.gov/`

[7] Mugan, J. and Kuipers, B., "Autonomous learning of high-level states and actions in continuous environments," *IEEE Trans. Autonomous Mental Development* **4**(1), 70–86 (2012).

[8] Dean, T. and Kanazawa, K., "A model for reasoning about persistence and causation," *Computational intelligence* **5**(2), 142–150 (1989).

[9] Mugan, J., "A developmental approach to learning causal models for cyber security," in [*SPIE Conference on Defense, Security, and Sensing, Machine Intelligence and Bio-inspired Computation: Theory and Applications VII*], (2013).

[10] Mugan, J., "Top-down abstraction learning using prediction as a supervisory signal," in [*AAAI Workshop on Learning Rich Representations from Low-Level Sensors*], (2013).

[11] Schneider, F. B., "Enforceable security policies," *ACM Transactions on Information and System Security (TISSEC)* **3**(1), 30–50 (2000).

[12] Jajodia, S., Noel, S., and OBerry, B., "Topological analysis of network attack vulnerability," *Managing Cyber Threats* , 247–266 (2005).