2018 IEEE Symposium on Security and Privacy Workshops

A Mathematical Modeling of Exploitations and Mitigation Techniques Using Set Theory

Rodrigo Branco*
Intel Corporation
Hillsboro, Oregon, USA
rodrigo.branco@intel.com

Kekai Hu*
Intel Corporation
Hillsboro, Oregon, USA
kekai.hu@intel.com

Henrique Kawakami*
Intel Corporation
Hillsboro, Oregon, USA
henrique.kawakami@intel.com

Ke Sun*
Intel Corporation
Hillsboro, Oregon, USA
ke.sun@intel.com

***Abstract*—One of the most challenging problems in computer security is formalization of vulnerabilities, exploits, mitigations and their relationship. In spite of various existing researches and theories, a mathematical model that can be used to quantitatively represent and analyze exploit complexity and mitigation effectiveness is still in absence.**

In this work, we introduce a novel way of modeling exploits and mitigation techniques with mathematical concepts from set theory and big O notation. The proposed model establishes formulaic relationships between exploit primitives and exploit objectives, and enables the quantitative evaluation of vulnerabilities and security features in a system. We demonstrate the application of this model with two real world mitigation techniques. It serves as the first step toward a comprehensive mathematical understanding and modeling of exploitations and mitigations, which will largely benefit and facilitate the practice of system security assessment.

Keywords. Computer Security, Language-theoretic Security, Exploit Modeling, Mitigation Effectiveness, Set Theory

I. Introduction

Just as the history of spears and shields, exploits and mitigations have been evolving competitively and interactively since the very beginning of computer security. The forms, types and approaches of exploits have expanded extensively from the simple classic stack overflow to more diverse and advanced ones like return-oriented programing (ROP) and call/jump-oriented programing (COP/JOP). While on the side of defense, mitigation techniques have also been developed and introduced with increasing quantity and sophistication: from the early-age stack canary and data execution prevention (DEP) to the more recent ones such as control flow guard (CFG) [1] by Microsoft, reuse attack protector (RAP) [2] by Grsecurity, and the control-flow enforcement technology (CET) [3] by Intel.

With the blowing up of the diversity and complexity of exploits and mitigations, it becomes more and more challenging to accurately describe the essence of a specific exploit and comprehensively evaluate the effectiveness of a certain mitigation with respect to the exploit. It remains unfinished work in the security field to establish a generic modeling for both exploits and mitigations to distill their essential aspects in a standardized manner so that the assessment can be made more straightforward, accurate, and consistent across different cases. Although there are well documented and categorized records such as common weakness enumeration (CWE) [4], such classifications are generally more descriptive and lacking of the necessary abstraction to extract the core essences from the superficial properties of an exploit.

A widely known and accepted concept on exploits in the security community is *weird machine* [5]. It clearly describes how a exploit happens and successfully predicts the bypassing of many existing exploitation countermeasures such as the control flow integrity (CFI) technologies [2], [6], [7]. While the definition of a *weird machine* pioneered the formalization of exploitations, more works are left blank on mathematical modeling and quantitatively analyzing exploitabilities of security vulnerabilities and the effectiveness of mitigation technologies.

In this work, we propose a mathematical modeling on formulating exploitations. The main contributions include:

1) Formal definitions of exploitation and mitigation. The notation and logic of set theory is applied to construct the representation of exploits and mitigations, with their primitives and attributes defined as set members and grouped as sub sets.
2) Apply the big O notation to mathematically describe the complexity of exploitations and the effectiveness of mitigation technologies.
3) Lay down the foundation of a general and practical modeling approach to map exploits and mitigations to an abstracted representation that accurately captures their essential properties, which can standardize and facilitate exploit-related narratives in security researches and practices.

II. Related Work

Prior to this work, there have been multiple related efforts in security research trying to model and theorize exploits and system states or behavior. The work by Sergey Bratus et al of Langsec [5], [8] introduces the concept of weird machine and considers exploits as constructive proofs of the presence of a weird machine. It uses this abstracted computational model to describe the famous examples in the history of exploits.

The work by Thomas Dullien [9], [10] considers a computer program as a finite state machine and points out the essence of exploitation is setting up, instantiating, and programming

*Authors are listed in alphabetical order

DOI 10.1109/SPW.2018.00050

the weird machine. It discusses the importance of program implementation and provides a theoretical understanding to distinguish unexploitable programs from exploitable ones.

In another work by Julien Vanegue on heap-related exploit modeling [11], [12], more detailed formal definitions are introduced to describe heap primitives and behaviors.

Despite the great value in these researches per se, they have yet to provide a general and actionable approach to map exploits and mitigations to a standard abstract representation, which will be covered by the scope of this work and discussed in detail in the following sections.

III. Exploit Modeling

In this section, we illustrate our set theory based mathematical modeling of security vulnerability and exploit.

A. Definitions and Terminologies

When a system has a security vulnerability, which is a flaw that can potentially undermine the system security [13], an attacker may be able to take advantage of it to maliciously manipulate the system. The process of an attacker manipulating a vulnerability is called exploit (verb). The word exploit as a noun is also used to refer a piece of software or a chunk of data or a sequence of commands that an attacker creates to make use of a security vulnerability. [14]

Not all of the security vulnerabilities can be used by an attacker to successfully launch an attack(i.e. they may not be exploitable). As one of the most challenging problems in exploit research, evaluating whether a vulnerability is exploitable depends on multiple conditions including but not limited to: the goal of an attacker, the ability provided by the vulnerability, the system status, and the mitigation implemented in the system, etc. In order to formalize the abstract representation of system properties, exploits and mitigations, a series of standard terms are defined here.

Definition III.1. An exploit primitive (EP) is an attack ability that an attacker can potentially achieve from a security vulnerability.

Each exploit primitive is composed of two elements: `type` and `property`. As suggested by their names, the `type` identifies the type of an exploit primitive which can be read, write or execute, while the `property` further describes the attack abilities associated with an exploit primitive type, such as location, timing, repeatability, etc. We define five major primitive properties in this model: arbitrary addresses, arbitrary content, arbitrary operation, arbitrary number of times and at arbitrary time.

All the security vulnerabilities and exploit technologies can be abstracted to exploit primitive representations. For example, a classical stack-based buffer overflow in a system with no protection can be represented with two exploit primitives: one write primitive with multiple bits on the stack and one execute primitive with arbitrary location on the stack.

Definition III.2. An exploit objective (EO) is the final goal that an attacker wants to achieve in a vulnerable system.

In a vulnerable system, an exploit objective can vary significantly based on different goals of different attacks. It can be either as simple as information leakage of a few memory bits or as complex as remote code execution.

With an exploit objective defined in a vulnerable system, the next step is to distinguish the exploitable vulnerabilities from the non-exploitable ones. Exploit condition is defined for this purpose.

Definition III.3. An exploit condition (EC) is the minimal required combination of exploit primitives in a vulnerable system to make this system exploitable to an exploit objective.

Exploit condition has three fundamental attributes:

- An exploit condition is always associated with an exploit objective, i.e., an exploit condition should not be considered as a fixed condition for all the systems in all scenarios, instead, it should be carefully specified with respect to an exploit objective. For example, with an exploit objective of information leakage, a read primitive with multiple bits at arbitrary location is definitely considered as exploitable while the same read primitive in the same system without other exploit primitives should be considered non-exploitable for an exploit objective of remote code execution.
- An exploit condition is the minimal requirement of an exploit. Thus, if any primitive in an exploit condition is removed, this exploit condition is not exploitable any more.
- One exploit objective can have many different exploit conditions. If and only if any of these exploit conditions is met, the exploit objective is exploitable. Take information leak as an example, both an exploit condition with an arbitrary read primitive and an exploit condition with a write and an execute primitives in the memory location that has read access to the target information would give the attacker the capability to steal the secret. Any one of these two exploit conditions makes the exploit objective information leak exploitable.

Definition III.4. An exploit complexity or exploit difficulty (ED) applies big O notation to quantitatively describe the upper bound of the time and cost for the attackers to exploit an exploit condition.

In a vulnerable system with no protection and no mitigation implementation, any exploit primitive is considered as O(1) complexity. By adding mitigation to the system, different levels of exploit complexities are introduced to corresponding exploit primitives. We will further elaborate this in Section IV.

B. Set Representation of Exploits

As defined in Definition III.1, an exploit primitive has two elements: `type` and `property`. We use a set T to denote all the exploit primitive types and a set P to denote all the exploit primitive properties as follow:

$$T = \{All\ EP\ types\} = \{Read, Write, Execute\} \quad (1)$$

$$\begin{aligned} P &= \{All\ EP\ properties\} \\ &= \{arbitrary\ addresses, arbitrary\ content, \\ &arbitrary\ operation, arbitrary\ number\ of\ times, \\ &at\ arbitrary\ time\} \end{aligned} \tag{2}$$

Thus, an exploit primitive can be represented as a combination of a type $t \in T$ and a property $p \in P$.

$$ep = \{t,\ p\} \tag{3}$$

With exploit primitive defined, an exploit condition is a set of exploit primitives.

$$\begin{aligned} ec &= \{ep_1, ep_2, \ldots, ep_n\} \\ &= \{\{t_1, p_1\}, \{t_2, p_2\}, \ldots, \{t_n, p_n\}\} \end{aligned} \tag{4}$$

Depending on the number of security vulnerabilities and their capabilities in a system, the exploitability E of a defined exploit objective in this system can be represented as a set of all the possible exploit conditions.

$$E_{eo} = \{ec_1, ec_2, \ldots, ec_n\} \tag{5}$$

Now, let us define the two major set operations in this model: set union and set subtraction.

When a new vulnerability is found for a certain exploit objective, new exploit primitives will be available for the attackers, thus new exploit conditions may be added to the system. We use set union to represent the increment of the overall exploitability E as follows.

$$E'_{eo} = E_{eo} \cup ec_{n+1} = \{ec_1, ec_2, \ldots, ec_n, ec_{n+1}\} \tag{6}$$

On the other hand, when one or more exploit conditions are removed or prevented in a system, the overall exploitability E will decrease. Set subtraction is used in this case to demonstrate the decline of the attacker's capability.

$$E'_{eo} = E_{eo} - ec_n = \{ec_1, ec_2, \ldots, ec_{n-1}\} \tag{7}$$

With all the exploit primitives that an attacker have in the system, for a certain exploit objective, if any of the exploit conditions is met, i.e., $E_{eo} \neq \emptyset$, this exploit objective is exploitable. To protect an exploit objective to be exploited, a mitigation need to block all the possible exploit conditions in the system. Otherwise, the mitigation doesn't fully protect the system and can be bypassed. If and only if $E_{eo} = \emptyset$, an exploit objective is not exploitable.

IV. Mitigation Modeling

In this section, we introduce the set theory and big O notation based modeling to abstract mitigations. This way, the mathematical mitigation modeling is standardized with the exploits, and quantitatively analysis of mitigation effectiveness is feasible.

A. Probabilistic and Deterministic Mitigations

In order to eliminate as much as possible the exploitability of unknown security vulnerabilities in real world systems, innumerous exploit mitigation techniques are designed and implemented in a variety of computing environments. After adopting one or more mitigations in a system, the system designer intends to protect one or more exploit objectives to be exploited by removing the attacker's abilities in the system, i.e., exploit primitives. If one or more exploit primitives are removed from an exploit condition by a mitigation, this exploit condition will not be met. For a specific exploit objective, if all of its exploit conditions have at least one exploit primitive removed from them, i.e., for all the $ec_i \in E, i = 1, 2, \ldots, n$, there is an $ep_j = 0, where\ ep_j \in ec_i$, this exploit objective will be non-exploitable with $E_{eo} = \emptyset$.

We propose a new taxonomy to classify these exploit mitigation techniques into two types: probabilistic mitigations and deterministic mitigations. The classification is based on their probabilities of preventing the exploitability of their target security vulnerabilities, as explained below.

Definition IV.1. A deterministic mitigation (DM) technique eliminates the exploitability of its target exploit primitives completely or reduces the chance of exploitation to a negligibly low probability that is not practical to exploit, under the assumption that the mitigation is implemented properly and the exploit is within its mitigation scope.

For instance, the NX bit, which is a technology used in CPUs to mark certain memory areas not executable, is an example of a deterministic mitigation. It completely blocks the exploit primitives of execute type in the marked memory ranges and malicious software can not break it since it is enforced at the CPU level.

In our mathematical model, we define that a deterministic mitigation adds $O(\infty)$ exploit complexity to a certain exploit primitive thus completely blocks it.

Definition IV.2. A probabilistic mitigation (PM) is a mitigation that increases the exploit complexity and reduces the successful rate of its target exploit primitives, but does not completely eliminate the exploit primitives.

A typical example of a probabilistic mitigation is address space layout randomization (ASLR). Although it can still be bypassed in many specific cases and does not fully remove the exploitability of memory corruption vulnerabilities, it reduces the possibility of exploiting it.

In our mathematical model, rather than defining a constant exploit complexity for all the probabilistic mitigations, we consider different exploit complexities to their target exploit primitives for different probabilistic mitigations. Take AES-128 as an example, assume that it takes 2^{128} tries for the worst case and 2^{127} tries for the average case to brute force the encryption key, we can use the average case exploit complexity $O(2^{127})$ to represent the exploit difficulty of AES-128. While in another example, address space layout randomization (ASLR) does not add too much protection in a system because it only takes $O(2^{16})$ exploit complexity to brute force attack a 16 bit ASLR.

B. Set Representation of Mitigation

From the security perspective, a computer system includes a set of valid mitigation technologies, each of which is either a deterministic mitigation or a probabilistic mitigation. An empty set of mitigations means the system has no protection. When a mitigation is applied to a system, it removes or reduces some of the attacker's abilities (exploit primitives), but other exploit primitives might not be affected at all. To generalize these exploit difficulty effects, we consider that a mitigation always adds exploit difficulties to all the exploit primitives in our model. For the target exploit primitives of the mitigation, it adds either $O(\infty)$ exploit complexity if it is a deterministic mitigation to this exploit primitive, or $O(n)$ exploit complexity if it is a probabilistic mitigation, where n represents the worst case exploit complexity to break this mitigation. For the exploit primitives that it does not add any complexity, and $O(1)$ is used to represent that there is no complexity change. We use the new notation $ep' = ep(ed)$ to represent an exploit primitive with its exploit difficulty. In a system that has no protection, no exploit complexity is added to any of the exploit primitives, hence, $ep' = ep(1)$.

This way, a mitigation can be represented in a set of its exploit primitives with the added exploit difficulties.

$$\begin{aligned} m = \{ep'_1(ed_1), ep'_2(ed_2), \ldots, ep'_n(ed_n)\} \\ where\ ed_i \in \{O(1), O(n), O(\infty)\} \end{aligned} \tag{8}$$

Now, let us define a mitigation operation M when a mitigation is applied to a system. Instead of simply removing one or more exploit primitives from an exploit condition, the exploit difficulties of all the exploit primitives in the system change, thus the exploit difficulty of a certain exploit condition will also change accordingly:

$$\begin{aligned} ec' &= ec\ M\ m \\ &= \{ep'_1(ed_1), ep'_2(ed_2), \ldots, ep'_n(ed_n)\} \\ &\quad M\ \{ep'_1(ed'_1), ep'_2(ed'_2), \ldots, ep'_n(ed'_n)\} \\ &= \{ep'_1(ed''_1), ep'_2(ed''_2), \ldots, ep'_n(ed''_n) \end{aligned} \tag{9}$$

In this operation, $ed_1, ed_2, \ldots, ed_n$ represent the exploit difficulties before the mitigation, $ed'_1, ed'_2, \ldots, ed'_n$ represent the exploit difficulties that are added by the mitigation, and $ed''_1, ed''_2, \ldots, ed''_n$ represent the exploit difficulties after the mitigation.

After applying a mitigation in a system, the new exploit condition ec' has three possible cases:

1) The new exploit condition ec' is not a valid exploit condition any more because at least one of its exploit primitives are removed from the exploit condition, i.e., for any of the $ep'_i \in ec'$ where $i = 1, 2, \ldots, n$, $ep'_i(ed''_i) = ep'_i(\infty)$.
2) The new exploit condition ec' is still exploitable but with higher exploit difficulty, i.e., for all of the $ep'_i \in ec'$ where $i = 1, 2, \ldots, n$, $ep'_i(ed''_i) \neq ep'_i(\infty)$ but some of the $ep'_i(ed''_i) > ep'_i(ed_i)$.
3) The new exploit condition ec' is still exploitable with the same exploit difficulty as before the mitigation, i.e., for all of the $ep'_i \in ec'$ where $i = 1, 2, \ldots, n$, $ep'_i(ed''_i) = ep'_i(ed_i)$.

Among these three cases, in contrast to the case 1 and 3 which are the straightforward yes or no cases, case 2 is the more complicated one depending on the number of exploit primitives that have $O(n)$ exploit difficulty. Take an exploit condition that has three exploit primitives as an example: assume that in a system that has no mitigation, all three exploit primitives have $O(1)$ exploit difficulty:

$$ec = \{ep'_1(1), ep'_2(1), ep'_3(1)\} \tag{10}$$

After implementing the mitigation, exploit difficulties for both ep_1 and ep_3 increase to $O(n)$:

$$ec = \{ep'_1(n_1), ep'_2(1), ep'_3(n_3)\} \tag{11}$$

For an attacker to achieve his exploit objective, he needs to bypass the protection on both ep_1 and ep_3 at the same time. In another word, his exploit complexity increases to $O(n_1) + O(n_3)$ eventually. With this example, we can see that the exploit difficulty of ec' in case 2 is the sum of $O(n_i)$ where i is the number of exploit primitives that has $ep'_i(ed_i) = ep'_i(n_i)$.

Now, let us look at the exploitations and mitigations at the system level to see how this model can be applied to system security evaluation. Consider a vulnerable system where an attacker can have five different exploit primitives that eventually meet three exploit conditions for a certain exploit objective. Assuming that there is no mitigation implemented in the initial state of the system, the exploitability of this exploit objective in the system can be represented as:

$$E_{eo} = \{ec_1(1), ec_2(1), ec_3(1)\} \tag{12}$$

Each exploit condition is a combination of exploit primitives, in this example, let us say ec_1 includes ep_1, ep_2, ec_2 includes ep_2, ep_3, ep_4, and ec_3 includes ep_1, ep_4, ep_5 as the following equation shows:

$$\begin{aligned} ec_1(1) = \{ep'_1(1), ep'_2(1)\} \\ ec_2(1) = \{ep'_2(1), ep'_3(1), ep'_4(1)\} \\ ec_3(1) = \{ep'_1(1), ep'_4(1), ep'_5(1)\} \end{aligned} \tag{13}$$

When a mitigation that can probabilistically mitigate ep_2, ep_4 and deterministically mitigate ep_5 is implemented in the system, it introduces extra exploit difficulties to the exploit primitives:

$$m = \{ep'_1(1), ep'_2(n_2), ep'_3(1), ep'_4(n_4), ep'_5(\infty)\} \tag{14}$$

Thus, the exploit difficulties of the exploit conditions change:

$$\begin{aligned} ec'_1(n_2) &= \{ep'_1(1), ep'_2(n_2)\} \\ ec'_2(n_2+n_4) &= \{ep'_2(n_2), ep'_3(1), ep'_4(n_4)\} \\ ec'_3(\infty) &= \{ep'_1(1), ep'_4(n_4), ep'_5(\infty)\} \end{aligned} \tag{15}$$

The overall exploitability E also changes:

$$E_{eo} = \{ec'_1(n_2), ec'_2(n_2+n_4), ec'_3(\infty)\} \tag{16}$$

In the new system, we can see that although the mitigation completely mitigates ec_3 and increases the exploit difficulty of ec_2 to $O(n_2+n_4)$, the exploit complexity of a certain exploit objective is always depending on the exploit condition that has minimum exploit difficulty, in this case ec_1 with exploit difficulty $O(n_2)$. Thus, the introduced mitigation is only a probabilistic mitigation to this exploit objective with $O(n_2)$ efficiency.

V. From Theory to Application

With the set theory modeling and big O notation defined in previous sections, real-world mitigation techniques can be abstracted into mathematical forms and evaluated quantitatively for its impact on system security robustness, with regard to specific exploit objectives, exploit conditions and exploit primitives. In this section, two classic mitigation technologies are used as examples, Control Flow Guard (CFG) by Microsoft and Control-flow Enforcement Technology (CET) by Intel, to demonstrate the application of the proposed model in evaluating effectiveness of mitigation techniques.

Control Flow Guard (CFG) is a mitigation technology to prevent control flow being redirected to unintended locations. It checks and validates if the target address of an indirect branch is a valid entry point before the branch can take place in the execution flow. In a system with CFG enabled, the number of legal target locations of an indirect branch is much smaller than an unprotected system. Thus, it is a probabilistic mitigation of call-oriented programming (COP) and jump-oriented programming (JOP) in the sense that, despite of not completely preventing such exploits, it largely reduces the availability of gadgets that can be used by COP and JOP.[15]

In the form of set representation, for the exploit objective of arbitrary code execution, COP and JOP need two basic exploit primitives to meet an exploit condition: overwrite the branch target address and execute at the target address, $ep_1 = Write, arbitrary\ content$ and $ep_2 = Exec, arbitrary\ addresses$.

$$ec(1) = \{ep_1(1), ep_2(1)\} \tag{17}$$

With CFG implemented, the arbitrary address in ep_2 is greatly reduced since only a small number of addresses are considered legal entries for control flow branch. Although CFG does not completely block ep_2, it makes it much harder and changes its exploit difficulty from $O(1)$ to $O(n_{CFG})$, where $O(n_{CFG})$ is the added complexity to carry out the COP and JOP type of control flow attack with legal gadgets only. Therefore, the exploit difficulty of this exploit condition also changes from $O(1)$ to $O(n_{CFG})$, and CFG introduces $O(n_{CFG})$ extra exploit complexity comparing to unmitigated case:

$$ec(n_{CFG}) = \{ep_1(1), ep_2(n_{CFG})\} \tag{18}$$

Besides CFG, another control flow integrity mitigation is Control-flow Enforcement Technology (CET) which contains two parts: indirect branch tracking (IBT) and shadow stack. The IBT part bears large similarity to CFG in terms of its purpose. It is a hardware-supported feature that inserts special labels to mark indirect branch targets as legal entry points and validates the label every time there is a indirect branch. Therefore IBT of CET is also a probabilistic mitigation of COP/JOP and has the effectiveness $O(n_{CET})$ similar to CFG. Both of them greatly reduce the possible gadgets rather than completely blocking the exploits.

The second part of CET is the shadow stack (SS), which pushes and pops the return address into a hardware-controlled stack independently from the active stack used by the process. It mitigates return-oriented programming (ROP) by checking the return address on the process stack with the shadow stack every time when a return instruction needs to be executed. Within its security premises, shadow stack can be considered as a deterministic mitigation for ROP (there could be exception cases such as implementation flaws that compromise partially or fully the mitigation, but in this work the mitigations are only considered by its designed figure of merits). In the proposed model, ROP must have an exploit primitive of overwrite the return address, $ep_1 = Write, arbitrary\ content$. With the mitigation of shadow stack, the exploit difficulty of this exploit primitive changes from $O(1)$ to $O(\infty)$, thus ROP attacks will be deterministically blocked:

$$ec(\infty) = \{ep_1(\infty), ep_2(1), \ldots, ep_n(1)\} \tag{19}$$

VI. Conclusion

In this paper, a set theory and big O notation based mathematical modeling of exploitations and mitigation techniques are introduced and assessed. The proposed model is designed to assist the evaluation of vulnerabilities and security features in a product, so that system designers can choose the optimal set of mitigations for a given set of exploits. On the other hand, the model can be also used by security analysts to find the optimal set of exploit primitives to achieve a given exploit goal. The current version of the model defines the theoretical framework and lay down the foundation for further elaboration and enrichment to make it more practical to be adopted in formal security analysis.

In summary, the use of this model can provide the following benefits: (1) Assist the evaluation of security features in a product, so that system designers can choose the optimal set of mitigations for a given set of exploits. (2) Find the optimal set of exploit primitives that are necessary to achieve a given exploit goal. (3) Provide a straightforward way of quantifying the impacts of a security-related bugs.

References

[1] Microsoft, “Control flow guard,” 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx

[2] P. Team, “Rap: Rip rop,” 2015, hackers to Hackers Conference. [Online]. Available: https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf

[3] Intel, “Control flow enforcement technology,” 2016. [Online]. Available: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[4] MITRE, “Common weakness enumeration.” [Online]. Available: https://cwe.mitre.org

[5] S. Bratus, M. Locasto, M. Patterson, L. Sassaman, and A. Shubina, “Exploit Programming: from Buffer Overflows to Weird Machines and Theory of Computation,” *USENIX ;login:*, Dec. 2011. [Online]. Available: http://langsec.org/papers/Bratus.pdf

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity,” in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS ’05. New York, NY, USA: ACM, 2005, pp. 340–353. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102165

[7] K. Hu, H. Chandrikakutty, R. Tessier, and T. Wolf, “Scalable hardware monitors to protect network processors from data plane attacks,” in *2013 IEEE Conference on Communications and Network Security (CNS)*, Oct 2013, pp. 314–322.

[8] J. Bangert, S. Bratus, R. Shapiro, and S. W. Smith, “The page-fault weird machine: Lessons in instruction-less computation,” in *Presented as part of the 7th USENIX Workshop on Offensive Technologies*. Washington, D.C.: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/woot13/workshop-program/presentation/Bangert

[9] T. F. Dullien, “Weird machines, exploitability, and provable unexploitability,” *IEEE Transactions on Emerging Topics in Computing*, vol. PP, no. 99, pp. 1–1, 2017.

[10] ——, “Exploitation and state machines,” 2011, infiltrate Offensive Security Conference. [Online]. Available: http://www.slideshare.net/scovetta/fundamentals-of-exploitationrevisited

[11] J. Vanegue, “The weird machines in proof-carrying code,” in *2014 IEEE Security and Privacy Workshops*, May 2014, pp. 209–213.

[12] ——, “Heap model for exploit systems,” 2015, iEEE Security and Privacy LangSec Workshop.

[13] I. Arce, “On the quality of exploit code: An evaluation of publicly available exploit code,” 2005, rSA Security Conference.

[14] C. Sarraute, “Automated attack planning,” *CoRR*, vol. abs/1307.7808, 2013. [Online]. Available: http://arxiv.org/abs/1307.7808

[15] M. Miller, “Modeling the exploitation and mitigation of memory safety vulnerabilities,” 2012, breakpoint Conference.