QUASAR: Quantitative Attack Space Analysis and Reasoning

Richard Skowyra
MIT Lincoln Laboratory
richard.skowyra@ll.mit.edu

Steven R. Gomez
MIT Lincoln Laboratory
steven.gomez@ll.mit.edu

David Bigelow
MIT Lincoln Laboratory
dbigelow@ll.mit.edu

James Landry
MIT Lincoln Laboratory
jwlandry@ll.mit.edu

Hamed Okhravi
MIT Lincoln Laboratory
hamed.okhravi@ll.mit.edu

ABSTRACT

Computer security has long been an arms race between attacks and defenses. While new defenses are proposed and built to stop specific vectors of attacks, novel, sophisticated attacks are devised by attackers to bypass them. This rapid cycle of defenses and attacks has made it difficult to strategically reason about the protection offered by each defensive technique, the coverage of a set of defenses, and possible new vectors of attack for which to design future defenses. In this work, we present QUASAR, a framework that systematically analyzes attacks and defenses at the granularity of the capabilities necessary to mount the attacks. We build a model of attacks in the memory corruption domain, and represent various prominent defenses in this domain. We demonstrate that QUASAR can be used to compare defenses at a fundamental level (*what* they do instead of *how* they do it), reason about the coverage of a defensive configuration, and hypothesize about possible new attack strategies. We show that of the top five hypothesized new attack strategies, in fact, four have been published in security venues over the past two years. We investigate the fifth hypothesized vector ourselves and demonstrate that it is, in fact, a viable vector of attack.

ACM Reference format:
Richard Skowyra, Steven R. Gomez, David Bigelow, James Landry, and Hamed Okhravi. 2017. QUASAR: Quantitative Attack Space Analysis and Reasoning. In *Proceedings of 2017 Annual Computer Security Applications Conference, Orlando, FL, Dec 2017 (ACSAC'17),* 11 pages.
DOI: 10.1145/3134600.3134633

1 INTRODUCTION

Computer security has long been an arms race. While defenders work diligently to propose and build new defenses that prevent various avenues of attack, attackers work just as hard to conceive and build new and clever means to bypass such defenses. The dynamics of this situation is exemplified by the state of attacks and defenses in the memory corruption domain. Although memory corruption attacks have been studied for more than four decades [3], increasingly complex defenses are continually built to guard against equally sophisticated attacks capable of bypassing the newest defenses. A few examples of such evolution in the community include: the deployment of W ⊕ X [26] followed by the advent of code reuse attacks to bypass it [34], code randomization and diversification techniques [23] followed by information leakage attacks [32, 38], fine-grained memory randomization [19] followed by just-in-time ROP (JIT ROP) attacks [37], control flow integrity (CFI) [2] followed by control Jujutsu [16] and control flow bending attacks [8], and code re-randomization [7] followed by data-oriented programming (DOP) [18].

This rapid defense/attack development cycle has made it difficult for both security planners and researchers to focus on the strategic view. On the one hand, defenses tend to focus on the narrow artifacts of an attack rather than on the attack's fundamental requirements, allowing motivated attackers to modify those artifacts and bypass the defense. On the other hand, security planners cannot easily understand the actual benefits offered by a defensive technique, compare incongruent defenses, or identify the types of attacks that are/are not covered by a particular defensive configuration.

There are some existing approaches for analyzing attacks and defenses, including low-level frameworks such as attack graphs [35] and high-level risk modeling and simulation frameworks [24, 41]. We argue that while valuable, these approaches are insufficient for strategic analysis of attacks, defense planning, and defense comparison. At the more detailed end of the scale, attack graphs analyze a network of computers at the granularity of individual software applications and their vulnerabilities. While such analysis provides high fidelity, the results are often only valid for a short duration because regular changes in the network such as (un)installation of an application, discovery of a new vulnerability, and even small modifications to the network topology or reachability, may significantly change or completely invalidate the analysis results. Moreover, attack graphs cannot easily compare various defenses since that would require solving the attack graph for different subsets of defenses, a problem that quickly becomes intractable as the number of possible defenses grows. In comparison, risk modeling and simulation approaches can analyze attack and defense interaction at a higher granularity, but such approaches usually require quantitative inputs such as attack arrival rates and defensive success

This material is based upon work supported by the Department of Defense under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense.

ACSAC'17, Orlando, FL. USA

DOI: 10.1145/3134600.3134633

probabilities. Such quantitative inputs are often not available or obtainable is a repeatable fashion in practice.

This work presents a framework called Quantitative Attack Space Analysis and Reasoning (QUASAR) to analyze attacks and defenses at a strategic level. QUASAR represents each attack class with a set of fundamental *capabilities* necessary for an attacker to successfully carry out that attack. Those capabilities may themselves rely on other finer-grained capabilities, combined using AND/OR operations in a graph-like structure, which we call an Attack Capability Graph (ACG).

For example, the attack type "code injection" requires that (1) the attacker knows the instruction set that is used on the target machine, and (2) the attacker has the capability to write to executable memory pages. In turn, the capability to write to executable memory pages requires (1) a write location, and (2) a write type. Each of these capabilities is in turn subdivided into further and more specific capabilities (*e.g.,* the write location may be the stack or the heap, and the write type may be a direct write or an overflow write). We represent a defense in QUASAR as a set of constraints on the ACG that disables some combination of attack capabilities. For example, stack canaries would put a constraint on the code injection subgraph that prevents the simultaneous usage of overflowing (for the write type) and stack (for the write location). However, with this defense, note that other combination of capabilities such as direct writing to stack are still possible. This approach allows us to represent defenses with a high degree of flexibility (unlike attack trees [21]) and can be viewed as a declarative way of specifying defenses: *what* defenses do instead of *how* they do it.

QUASAR can be used in defense planning to compare various defenses at a fundamental level and to analyze the coverage of a defensive configuration. It can also be used as a research planning tool to identify the capabilities most prevalently used in attacks (the *critical capabilities*), which currently lack coverage from existing defenses. Finally, it can be used to anticipate new high-impact attack strategies that attackers may begin to employ once effective defenses for current attacks are deployed.

To demonstrate these use cases, we create the entire ACG for the memory corruption domain, capturing all publicly disclosed attack types and their capabilities from 1972 to 2017, resulting in 140 fundamental attack capabilities. We also represent 20 prominent memory protection defenses including control-flow integrity [2], code pointer integrity [22], Readactor [11], TASR [7], binary stirring [40], ILR [17], as well as widely deployed defenses such as DEP [26], stack canaries [10], and ASLR [27]. Naïvly calculating defense coverage and attack criticality in such a large model quickly becomes intractable because an NP-hard problem must be solved an exponential number of times. Instead, we leverage a recent development in the formal methods community, called #SAT, that can quantify these measures without repeatedly solving the full satisfiability problem.

This approach allows us to use QUASAR to meaningfully compare several seemingly-incongruent defenses to code reuse attacks against each other. Furthermore, QUASAR allows us to identify the top high-impact attack strategies that attackers may next target given the state of the best proposed defenses today. As validation for this latter use, during the time that this project has been under development, four of the top five high-impact attack strategies identified by QUASAR have been published in various security venues. We briefly study the fifth high-impact strategy (software-based malicious DMA attacks) ourselves and construct a proof-of-concept attack against SFTP to illustrate its practicality and impact, and furthermore show that the opportunities for such attacks are prevalent in various popular server applications.

Our contributions are as follows:

- We propose a framework to analyze attacks and defenses at the granularity of attack capabilities, not too low-level to impede strategic reasoning and not too high-level to require hard-to-quantify values.
- We build an entire attack capability graph (ACG) for the memory corruption domain, comprising 140 fundamental attack capabilities, and also represent various prominent memory protection defenses.
- We leverage recent advancements from the formal methods community to compute informative measures such as defense coverage and attack capability criticality in the memory corruption ACG.
- We demonstrate how ACGs can be used to specify the precise impact of each defense and compare defenses.
- We identify the top five new high-impact attack strategies that we anticipate attackers use in response to recent defenses proposed in the community. We further illustrate that four of these attack techniques have in fact been published in the past 2 years.
- We investigate the fifth high-impact attack strategy (software-based malicious DMA), build an example exploit against a real-world application, and otherwise study its applicability and prevalence.

Our results indicate that analysis of attacks and defenses at the granularity of attack capabilities allow better strategic reasoning for defense planners and researchers alike.

2 MODELING ATTACKS AND DEFENSES

QUASAR is a framework that helps quantify how critical particular attack techniques might be for attackers, and measures the coverage of defenses with respect to these attack techniques. To compute these measures, it is necessary to produce an encoding of existing knowledge about attacks and defenses in a specific security domain. We call this a *knowledge base*, which is manually curated and constructed by security experts, and ultimately represented as a set of logical expressions. This allows QUASAR to leverage #SAT solving for its quantitative measures, which is the counting variant of Boolean Satisfiability. At a high level, a QUASAR model compiles to Boolean formulas whose solutions correspond to successful attack methods. By counting the number of solutions in the presence of specific defenses, we can create measures that enable defenses to be analyzed and compared. This section will cover the model creation and compilation process, while Sections 3 and 4 will go into detail on measures for comparison and gap analysis.

2.1 Attack Capabilities

In order to compare defenses both against each other, and against the space of attacker strategies, we need to choose a granularity

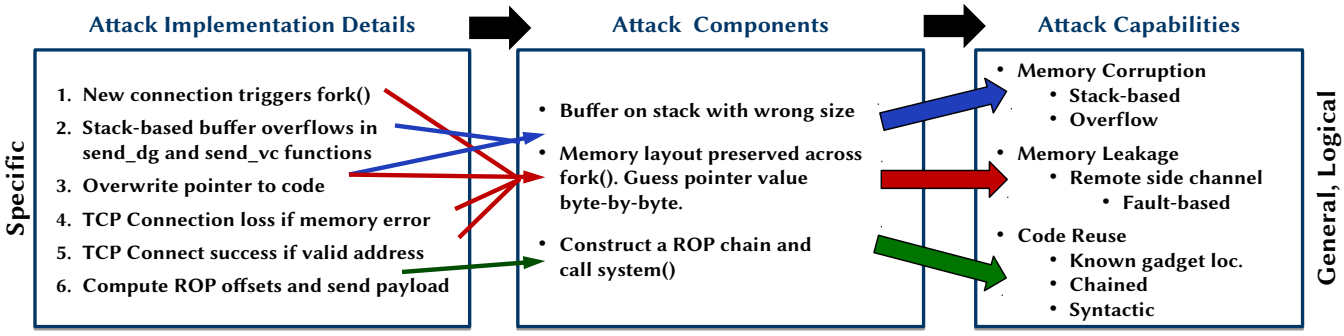

Figure 1: Linux `getaddrinfo` attack

of abstraction that captures the impact of defenses on attacks, but is neither so specific that conclusions are not generalizable (*e.g.,* depending on specific software versions) nor so abstract that it requires substantial parameterization from unavailable datasets (*e.g.,* Monte Carlo simulations). We elected to use an attack tree-like representation of attacker *capabilities*, which represents features of the target system that must be present in order for an attack *method* to succeed despite the presence of deployed defenses.

Consider the example in Figure 1, which describes the `getaddrinfo` attack disclosed in February 2016 [1] against Linux's libc library. This attack enabled malicious DNS servers to achieve remote code execution on vulnerable systems. The left-most box contains a summary of detailed implementation data: specific function names, specific memory offsets, and specific side channels. This level of granularity is helpful for understanding the artifacts of an attack, but is tightly coupled to specific system configurations. Upon abstraction into large-scale components (center box), these details resolve themselves into mechanisms for memory corruption, memory leakage, and code reuse. The right-most box depicts what we call *attack capabilities*, which are a formalized, taxonomic approach to characterizing an attack. The `getaddrinfo` attack uses, for example, memory corruption on the stack via an overflow.

This level of granularity is helpful for two reasons. First, the right-most depiction of the attack is agnostic of specific implementation details. It focuses purely on *what* features the attacker requires, not *how* they are implemented. This enables reasoning about large classes of attack and how defenses may impact their critical requirements, rather than analysis of artifacts that attackers could potentially bypass. Second, because attack capabilities are shared across many implementations of attacks, a relatively small number of such capabilities are needed to fully capture all disclosed attack methods. This allows QUASAR to avoid scalability challenges that would naturally arise in finer-grained modeling approaches like attack graphs.

Attacker capabilities are dependency relationships, beginning with a high-level goal (*e.g.,* "control flow hijacking") and continuing into increasingly-refined sub-goals. Each attack capability can have either an "All-Of" or "Any-Of" relationship with other capabilities on which it depends, or it can be an atom with no further dependencies. This can be represented in an *attack capability graph* (ACG). For example, consider the ACG depicted in Figure 2, which is a small subgraph of the entire memory corruption ACG, capturing code injection attacks. The top-level goal, *Control_Flow_Hijacking*, has an Any-Of relationship with its sole dependency, *Code_Injection*. That has an All-Of relationship with both the *Write_To_Executable_Memory* and *Instruction_Set_Known* capabilities; both dependencies must be satisfied in order for the attacker to be able to use code injection. These dependencies have their own dependencies, which must themselves be satisfied. Capabilities that represent the lowest level units of an attack (*e.g.,* *Overflow* in the example above) are considered attack *atoms* and are not dependent on other capabilities; however, such atoms can easily be extended with new requirements if it should become necessary by the identification of a new subdivision in capabilities or defenses. In addition to extending atoms, an ACG can be further expanded by adding new high-level capabilities and dependencies. Control-Flow Hijacking, for example, can also be accomplished via Code Reuse (*e.g.,* Return-Oriented Programming). A dependency model can be added to the existing Any-Of dependency for *Code_Reuse*, and form a new subgraph of the larger ACG.

The ACG representation makes model construction easy for human operators, but is otherwise equivalent to a logical formulation of attacker capabilities that is amenable to automated analysis by QUASAR. Formally, an attack capability is defined as a propositional logic formula with one of the following structures:

$$\begin{aligned} &\text{All-Of: } a_n \rightarrow (a_i \wedge \ldots \wedge a_j) \\ &\text{Any-Of: } a_n \rightarrow (a_i \vee \ldots \vee a_j) \\ &\text{Atom: } a_n \end{aligned} \tag{1}$$

where:

- a_n is the name of the attack capability. This atom may be constrained by defenses, and is only available if its requirements (if any) are met.
- $a_i \ldots a_j$ are the attack capabilities on which a_n depends.

Note that the ACG could be represented in Datalog, but we choose to use propositional logic notation because Datalog cannot be used to encode the entire model. Specifically, defenses may impose constraints that negate particular attack capabilities. To create a unified attack capability model, a single formula ϕ_{atk} is created as the conjunction of all attack capabilities, along with the

assertion of the top-level goal:

$$\chi_{atk} = \bigwedge_{a \in A} a$$
$$\phi_{atk} = \chi_{atk} \wedge a_g \tag{2}$$

where:

- A is the set of all attack capabilities
- χ_{atk} is the entire attack capability graph
- a_g is the top-level goal of the attack capability graph

This structure creates a formula with a key property: it is satisfiable only if there exists a set of attack capabilities which permit the top-level goal a_g to satisfy all of its dependencies. Thus, every solution to the formula is a unique attack method, by virtue of being a unique set of truth values that satisfy the formula and make a_g achievable by the attacker.

2.2 Defenses

Defenses in the knowledge base serve to constrain an attacker's ability to use one or more capabilities. In general, defenses are encoded in a manner similar to attack capabilities; however, where attack capabilities must be attack atoms or have dependencies on other capabilities, defenses can arbitrarily constrain capabilities. This flexibility is a novel contribution of our work and does not exist in similar approaches, such as attack/defense trees [21].

For example, Data Execution Prevention (DEP) constrains an attacker's ability to write to executable memory. We can encode DEP as assuring that the attack capability *Write_To_Executable_Memory* is disabled as far as any upstream capabilities are concerned, regardless of whether the dependencies of *Write_To_Executable_Memory* are satisfied. A defense is formalized as a logical formula of the following structure:

$$d_n \wedge \phi_r \rightarrow \phi_a \tag{3}$$

where:

- d_n is the name of the defense. Note that the constraint it imposes over the attack space is enforced only if d_n is asserted.
- ϕ_r is a requirement that the defense has over some set of system environment atoms, such as being limited to deployment on certain platforms (*e.g.,* Linux vs. Windows).
- ϕ_a is the constraint imposed by the defense on available attack capabilities. This may be an arbitrary logical constraint, and is not limited to simple negation.

This structure has two useful properties. First, defenses can be added to the model without actually deploying them (*i.e.,* enforcing their constraint). This is useful, as it enables QUASAR to decouple model creation from model querying. That is, we can efficiently count formula solutions (and thus the number of viable attack methods) for any combination of defenses, which need not be chosen in advance at model compilation time.

Second, defenses impose *arbitrary* constraints over attack capabilities, which is particularly critical in the memory corruption domain. Coarse-grained control-flow integrity, for example, does not directly remove any capabilities from an attacker's arsenal but instead limits their combination by imposing a policy: dereferencing of a return address may only go to a call site, and dereferencing an indirect branch may only go to a function prologue.

Because defenses can impose arbitrary constraints which may be difficult to write by hand, we developed a small domain-specific language to express constraints more compactly than by using boolean expressions over capabilities. These are automatically expanded by QUASAR into the full Boolean formula. For example, we represent coarse-grained control flow integrity in the following way:

```
dcap Control_Flow_Integrity_Coarse_Grained assures
 (Function_Return.Dereference_Event implies
(Entry_Point.Code_Pointer_Dereference I
{Call_Site.Entry_Point}))
and
(Explicit_Dereference.Dereference_Event implies
(Entry_Point.Code_Pointer_Dereference I
{Function.Entry_Point}))
```

The language has two features to note beyond simple Boolean operations: postfix-matched names and set operations. The former servers to disambiguate attack capabilities which may have the same name. `Stack` appears in several locations of the attack capability graph, for example, with different contexts and semantics. Rather than demand unique names, defenses can specify which capability is constrained by postfixing it with a chain of its parents, separated by dots. For example, `Function_Return.Dereference_Event` refers to the `Function_Return` attack capability with `Dereference_Event` as its parent. QUASAR will not compile the model if any ambiguity is present, which must be resolved by extending the length of the postfix until it is unique.

The second feature is the use of two simple set operations over the children of an attack capability: intersection (`I`) and complement (`C`). Each non-terminal attack capability can be viewed as the name of a set whose members are its children. Operations over this set can allow defenses to more clearly and easily constrain which children of that attack capability are constrained or available. For example, (`Entry_Point.Code_Pointer_Dereference I Call_Site.Entry_Point`)) refers to the `Entry_Point` set, which denotes the means by which a code pointer de-reference can point to the entry point to a code region. Its members are `Function`, `Call_Site`, `Aligned_Instruction` and `Unaligned_Instruction`. The intersection of `Entry_Point` with `Call_Site` restricts the attacker from using any other entry point but a call site.

This notation allows defenses to avoid explicitly listing which alternatives they restrict. It allows more terse, readable defenses, as well as somewhat insulating defenses from needing to be rewritten if the attacker capability graph is updated to include another alternative in an already-restricted set.

Once all defenses have been defined, they are conjoined into a single Boolean formula, which is itself conjoined with the space of attack capabilities captured by ϕ_{atk}:

$$\phi_{def} = \bigwedge_{d \in D} d$$
$$\phi_{full} = \phi_{def} \wedge \phi_{atk} \tag{4}$$

Using the formalism defined above, we created an ACG for memory corruption, comprising 140 fundamental attack capabilities.

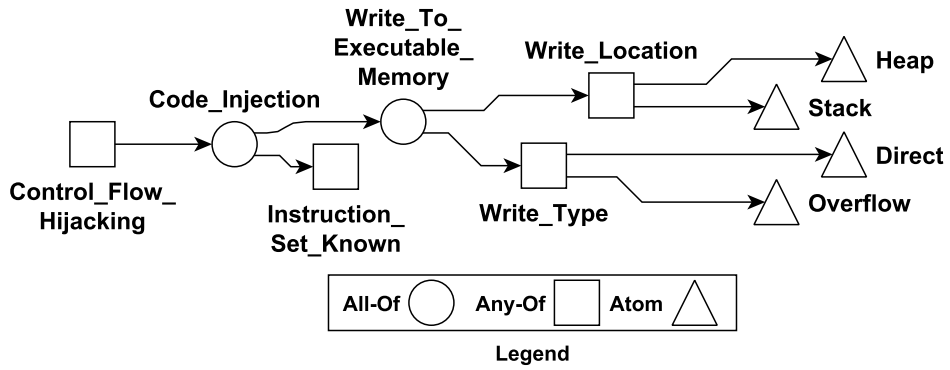

Figure 2: Code Injection Attack Capability Graph

All publicly disclosed attack methods (*e.g.,* Return-Oriented Programming (ROP), Ret-to-Libc, Counterfeit Object-Oriented Programming, JIT-ROP) are included as sets of capabilities that may be chosen by an attacker. This ACG is shown in Figure 3.

We also modeled twenty prominent defenses, including control-flow integrity [2], code pointer integrity [22], Readactor [11], TASR [7], binary stirring [40], ILR [17], as well as widely deployed defenses such as DEP [26], stack canaries [10], and ASLR [27].

2.3 Automated Solution Counting via #SAT

The formula ϕ_{full} representing the model of defenses and attack strategies has an important property: every solution to that formula constitutes a successful attack method, and every successful attack method constitutes a solution to the formula. This means that the ability to count formula solutions enables the counting of attack methods. Furthermore, defenses are not explicitly enabled. As logical implications, the constraint they impose on the selection of attack capabilities are enforced only if the defense itself is asserted. If the number of solutions to the formula can be counted given some set of defense literals that are enabled or disabled, however, we can perform quantitative reasoning about how many unique attack methods are possible in presence of a specific defense (or set of defenses).

Unfortunately, traditional SAT solvers cannot be usefully leveraged for counting the number of unique solutions to a formula. SAT solvers rapidly find and return individual solutions, whereas we wish to count (but not necessarily enumerate) the total number of formula solutions.

A technique known as #SAT, or propositional model counting, does provide the ability to count formula solutions [6]. Specifically, an approach known as "Knowledge Compilation" transforms a Boolean formula into an efficient representation known as Smoothed Deterministic Decomposable Negation Normal Form (SDDNNF) [12, 13]. This representation has several useful properties.

First, although the compilation step is itself NP-Hard, queries against the SDDNNF are performed as linear-time graph traversals. Because our graph is guaranteed to be polynomial with respect to number of variables in the input formula, the expensive computational step need only be performed upon model creation or modification, rather than during the solution counting phase.

Second, the SDDNNF allows solution counting to be implemented as a function $G(S)$ which takes a constraint set as a parameter S, and returns the solution count under those constraints. Constraints in this context are a set of truth values for atoms in the formula. This can be used to conduct an analysis of attack methods conditioned on a specific set of defenses being asserted (*i.e.,* deployed) and all others negated. We leverage this in Section 3 to compare defenses by measuring their coverage over the space of attack methods.

Finally, the partial derivative of the solution counting function can be computed with respect to each atom of the formula. This enables us to compute, for every atom, the number of solutions when that atom is either asserted or negated. Computation for each atom can be done simultaneously, in a second linear time graph traversal. We use this capability in Section 4 to identify future high-impact attack capabilities.

For this work, we used the C2D [14] #SAT solver, which outputs a formula in SDDNNF. Despite having approximately 500 variables covering attack capabilities and defenses, compilation times never exceeded 500ms on a 2.1GHz Intel Xeon processor with 32GB of RAM.

3 COMPARING DEFENSES

3.1 Comparing Defense Constraints

Representing memory corruption defenses as logical constraints on attack capabilities can be a useful way to identify comparable or — with respect to our model — equivalent defenses. For example, consider two anti-ROP defenses:

- ILR [17] is a virtualization-based defense which randomizes instruction locations in a process.
- Binary Stirring [40] is a loader-based defense which reorders and re-writes native code to be semantically equivalent on the basic block level.

Each of these defenses rely on very different mechanisms, working at varying levels of abstraction and varying points in the program's lifecycle. How, then, may they be compared against each other? When distilled to the constraints that they impose on available attack capabilities, however, we realize that all three defenses are logically equivalent:

$$\neg Memory\_Layout\_Known \rightarrow \neg Syntactic\_Gadgets \quad (5)$$

Figure 3: Attack capability graph for memory corruption attack methods.

That is, all three defenses apply only when the memory layout is not known to an attacker. In that case, they prevent the attacker from using syntactic (*i.e.,* traditional ROP) gadgets. It is also important to note that we are considering the "ideal" version of each defense as claimed by the authors; the actual implementation may have other weaknesses not included in this formulation.

By modeling defenses at this level, we gain the ability to identify comparable or logically equivalent defenses. This enables, for example, identification of common threat model assumptions (*e.g.,* attackers not knowing memory layout) that might indicate a shared point of weakness across several defenses. It also allows researchers or defense planners to avoid spending resources on developing or deploying a defense which provides no added protection beyond already existing techniques.

3.2 Comparing Defense Coverage

Another strategy for comparing defenses to one another is through analysis of the degree to which they constrain attack methods. Recall that defenses in QUASAR are represented as logical constraints over attack capabilities. These constraints limit what capabilities can be simultaneously selected as part of an attack method. By counting how many distinct successful attack methods are possible in the presence of a deployed defense, we can compute a measure of that defense's coverage over the space of attack methods. Specifically, for each defense in the model we can compute the following Defense Coverage measure:

$$\forall d \in D: \ \text{Coverage}(d) = 1 - G(S_d)G(S_{off})^{-1} \tag{6}$$

where:

- D is the set of literals corresponding to all defenses included in the model.
- G is the solution counting function which takes a set of constraints and returns the number of formula solutions under those constraints.
- $S_d = \{d\} \cup \{\neg d' \in D | d' \neq d\}$ is a constraint set in which the defense d is enabled, and all other defenses are disabled.
- $S_{off} = \{\neg d' \in D\}$ is a constraint set in which all defenses are disabled.

That is, defense coverage is a proportional measure which compares the number of attack methods available when that defense is deployed, to those available when no defenses are deployed. This is normalized to be in the $[0, 1]$ interval, where a higher value corresponds to higher coverage, meaning a lower number of successful attack methods. With respect to our model of defenses and attack capabilities, coverage never exceeds approximately 0.4 for any single existing defense. This is because no single defense can stop the majority of attack methods. It is possible that for strong defenses, the remaining attack methods may be qualitatively harder. This, however, is difficult to quantify and is currently not a part of our analysis. The approach used to compare single defenses can also be used to compare sets of defenses against each other by enabling them in S_d.

For example, consider two memory corruption defenses: Readactor [11] and Stack Cookies [4]. The constraints imposed on the attack capabilities for each defense are shown in Table 1, as are their defensive coverages. Readactor is a recent memory corruption defense that relies on fine-grained randomization and non-executable memory and with respect to our attack capability model, it constrains attack methods such that: the text segment of a process is unreadable to an attacker, the victim's memory image must be known (*i.e.,* an attacker's local copy cannot be used to identify gadget locations), the only indirect branches are trampolines (rather than direct pointers to the branch target), and code reuse on the semantic level cannot rely on counterfeit objects. Due to constraining many attack capabilities, often forcing the attacker to alternatives which themselves rely on complex dependencies, Readactor provides a high defense coverage. Attacks are still possible, but are generally restricted to those with many complex dependencies or which rely on multi-stage attacks [29].

In comparison, stack canaries protect against buffer overflows on the stack, assuming no memory disclosures are present. This is captured in our defense constraint. If the memory layout of the victim is unknown, and the memory corruption happens via an overflow on the stack, then the attacker's corruption of process control data cannot include corruption of return addresses. This is a much weaker constraint than Readactor, as is clear when comparing their respective defense coverages.

4 NEW HIGH-IMPACT ATTACKS

4.1 Attack Criticality

In addition to its ability to compare defenses against one another, QUASAR can also be used to compare attack capabilities against each other. The intuition behind our attack criticality analysis is this: for a given set of deployed defenses, there is a set of attack methods that will succeed despite the presence of those defenses. These attack methods rely upon specific capabilities, and some of those capabilities may be leveraged more than others. Thus, attack criticality analysis is a per-capability measurement showing what proportion of successful attack methods rely on that capability being available. Computation of attack criticality is based on the partial derivative of the solution-counting formula:

$$\forall a \in A: \ Criticality(a, D) = \frac{\partial G(D)}{\partial a} G(D)^{-1} \tag{7}$$

where:

- A is the set of all attack capabilities
- G is the solution counting function which takes a set of constraints and returns the number of formula solutions under those constraints.
- D is a constraint set denoting enabled defenses.

Computing the partial derivative of $G(D)$ with respect to attack capability a returns two values: the number of solutions when a is enabled and the number when a is disabled [13]. For this analysis, we are interested in the case where a is enabled, meaning it is used by successful attack methods. This is then divided by the total number of attack methods given the deployed defenses, creating a proportional attack criticality score in the $[0, 1]$ interval. High attack criticality indicates attack capabilities which are used by many successful attack methods, as well as those which have very few of their own dependencies and are thus more generalizable. A lower criticality indicates attack capabilities that are blocked or

Table 1: Comparing Defense Coverage

Defense	Constraint	Coverage
Readactor	$Memory\_Accessed \rightarrow (Data \wedge \neg Text) \wedge Memory\_Layout\_Known \rightarrow (Memory\_Image\_Known \wedge \neg Disk\_Image\_Known) \wedge Function\_Pointer\_Type \rightarrow (Trampoline \wedge \neg Direct) \wedge Return\_Address\_Type \rightarrow (Trampoline \wedge \neg Direct) \wedge Semantic\_Abstraction\_Level \rightarrow (\neg Counterfeit\_Objects \wedge Counterfeit\_Procedures)$	0.32
Stack Canaries	$\neg Memory\_Layout\_Known \rightarrow (Overflow \rightarrow Process\_Control\_Data \rightarrow (\neg Return\_Address \wedge (Exception\_Handler \vee Heap\_Metadata \vee Virtual\_Function\_Table)))$	0.046

limited by deployed defenses, as well as those capabilities which have a large number of required dependencies.

Attack criticality can be used to identify, for some set of deployed defenses, the highest-impact attack capabilities that can be used against that defensive posture. That is, these are the areas where attackers are most incentivized to focus their attacks, and thus in turn where defense planners and researchers should prioritize mitigations.

4.2 Anticipating High-Impact Attacks

In April 2015 we conducted an experiment to test the real-world utility of attack criticality measures. We further added attack capabilities to our model which were hypothesized in the literature, but had not necessarily been proved feasible in practice. We then enabled a combination of defenses that are ubiquitously deployed (DEP and ASLR), as well as the following state-of-the-art defenses: Timely Address Space Re-randomization (TASR) [7], control-flow integrity [2], Readactor [11], code-pointer integrity (CPI) [22], and gadget-smashing defenses [17, 25, 40].

This defensive configuration is likely to be impractical in real-world, due to both imposed overhead and potential incompatibilities among defenses. It does, however, represent an ideal best case for defenders and worst case for attacks. The highest-criticality capabilities would thus represent where attackers are most incentivized to concentrate their development efforts.

We then computed attack criticality against this defensive posture, and identified the five highest-criticality hypothesized capabilities as well as the defenses they would bypass, were they to be realized. The results are depicted in Table 2.

One month after our initial analysis, Schuster *et al.* [31] released an attack against C++ programs using object semantics to conduct code reuse attacks without relying on any ROP gadgets being present. Over the next several months, as documented in Table 2, four of the five high-impact attack capabilities were developed and published either in the academic literature or as proof-of-concept exploits. The final high-value attack capability that we identified in the top five is a software-based approach to carrying out Direct Memory Access attacks, bypassing the memory permissions enforced by the Memory Management Unit. Hardware variants of this attack have already been published (*e.g.,* attacks against Thunderbolt [33]), but we primarily consider software-based attacks. In order to demstrate the viability of this capability, we created a proof-of-concept attack using software-based forged DMA requests to bypass memory protections.

5 FORGED DMA ATTACKS

As described in Section 4, QUASAR identified the bypassing of memory permissions in software as a high-criticality, undefended, and as-yet unused attack capability in the memory corruption domain. The key enabler is found in the "Access_Method" portion of the ACG in Figure 3, where memory is reachable by either Direct Memory Access (DMA) or the Memory Management Unit (MMU). Memory permissions are enforced by the MMU and our model contains the implicit assumption that the MMU is handing those permissions appropriately and correctly; therefore, bypassing memory permissions must instead be achieved by DMA. A slightly different model might split the MMU into multiple categories to highlight other potential access vectors and capabilities, but the overall capability of bypassing memory permissions in software would remain valid and of high criticality.

5.1 DMA as an Attack Vector

Direct Memory Access (DMA) has long been recognized as an alternate path into memory for both malicious [5] and benign [9] purposes. DMA is an architectural feature that removes the CPU from the critical path when transferring data between memory and other external hardware devices such as disk drives and network interface cards. This has the dual benefit of both speeding up the data transfer, and allowing the CPU to perform useful work while waiting for the transfer to finish. However, this type of access bypasses the Memory Management Unit, which translates virtual addresses to physical addresses and ensures that page permissions are enforced. Attacks that exploit this bypass method have been carried out in both physical and virtualized environments, either by exploiting I/O controllers and firmware in externally connected devices (*e.g.,* USB, Firewire, Thunderbolt) or simply by building malicious hardware and physically connecting it to a system [30]. The need to protect against this type of malicious DMA attack led, in part, to the introduction of the IOMMU (Input/Output MMU) in x86 and other architectures.

If we assume that the MMU is properly configured, an attcker must bypass the MMU to be able to bypass memory permissions in software. However, DMA is difficult to perform in a userspace context. An unprivileged userspace process cannot issue a DMA request directly to hardware, and must therefore convince the kernel to make such a request on its behalf. Unfortunately, such a userspace-accessible functionality does exist that can be exploited to trick the kernel into executing a malicious DMA request on behalf of an unprivileged process (the "confused deputy" problem).

Table 2: Anticipating Development of Attack Capabilities

Attack Capability	Attack Criticality	Defenses Bypassed	First Appearance in the Wild
Partial Pointer Overwrites	0.663	TASR	Nov 2016 [15]
Software-Based DMA Attacks	0.651	DEP, Readactor, CPI	**Section 5**
Data-based memory disclosure	0.595	ASLR, TASR, CPI	Mar 2016 [18]
Code reuse inside control-flow graph	0.543	CFI	July 2015 [8]
Semantic code reuse	0.531	Gadget Smashing	May 2015 [31]

We dub such an attack a Forged DMA (FDMA) attack (not to be confused with hardware-based malicious DMA attacks [5, 30]). The primary example we will use here is the `O_DIRECT` flag in Linux, but other examples exist such as OpenCL's `CL_MEM_USE_HOST_PTR` flag.

The `O_DIRECT` flag is used as an argument to block I/O devices to transfer data directly between userspace buffers and the device without going through the kernel. In an FDMA attack, a remote attacker can corrupt a program's call to `open()` which uses `O_DIRECT` or has a flags argument that can be modified by the attacker. The attacker then subverts a `write()` call to point to an area of memory he/she does not have permissions to access (*e.g.,* to read a code page marked as executable-only by a defense like Readactor [11]). Thus, such an attack can bypass memory permissions because the memory access is serviced as a DMA request, rather than an MMU access. At a high level, such an attack is equivalent to the attacker being able to run the following code:

```
1  fd = open("code.bin", (O_DIRECT | O_WRONLY));
2  write(fd, page_aligned_program_code_pointer, 4096);
3  close(fd)
4  fd = open("code.bin", O_RDONLY);
5  read(fd, outgoing_buffer, 4096);
```

This copies a portion of the program code to a file regardless of memory permissions, and then re-reads the contents of the file which has not preserved the original permissions.

5.2 Example Exploit against SFTP

As a proof-of-concept, we have developed an exploit making use of the FDMA attack against the standard SFTP program, which is part of the OpenSSH suite of applications. The attack targets two functions that are executed when the client uploads a file: `process_open` and `process_write`. At a high-level these functions implement the following functionality (simplified for illustration purposes):

```
1  int process_open() {
2      int fd, flags, mode;
3      char *name;
4      fd = open(name, flags, mode);
5      handle_fd(fd);
6  }
7  int process_write() {
8      int fd, data_length;
9      char *data;
10     write(fd, data, data_length);
11 }
```

The attack is carried out as follows.

- We first leak stack (data memory) to find a code pointer.
- We convert the code pointer into a page-aligned code pointer.
- We connect to the SFTP server and send a file, *code.bin*.
- The `process_open()` function is called to write the file to disk. We corrupt the `flags` variable to include `O_DIRECT` before the call to `open()`.
- The `process_write()` function is called shortly afterwards. We corrupt the data variable to point to the calculated code address and the length to be block aligned. This causes code from the running SFTP process to be written to the specified file regardless of memory permissions.
- We retrieve the *code.bin* file by corrupting a read request.

At the end of this process, the file *code.bin* contains code from the SFTP process, having been retrieved without regard to the memory permissions of that piece of memory. For the sake of brevity, we do not include the entire exploit payload in the paper, but we have successfully tested this exploit against a system protected by DEP, ASLR, and Readactor.

5.3 Evidence of Exploitable Real-World Conditions

The `O_DIRECT` flag has been available on Linux since kernel version 2.4.10 and, as previously mentioned, is used for block device I/O. The option is used for certain specialized applications such as databases, which handle their own caching and I/O scheduling, but *can* be used for any appropriate file. From the attacker's standpoint, this attack is thus best suited to situations where a disk can be used as an intermediate location to store and retrieve a memory dump via a file, but the attack would also work for other block devices where `O_DIRECT` is applicable. Attacks can be carried out on file I/O that already make use of the `O_DIRECT` flag by corrupting the filename and the data buffer; alternately, attacks can make use of any `open()` call where the flags are specified in a variable rather than directly coded by also corrupting that flags variable in memory.

We investigated the usage of `O_DIRECT` and of variable-resident flags in a variety of popular webservers (Apache, Nginx, lighttpd, Boa, AOLserver, OpenSSH, and Squid) and database managers (MongoDB, MySQL, PostgreSQL, Redis, SQLite, memcached, MariaDB, Hypertable, and Firebird), both of which make good targets when file I/O is involved. We discovered that use of `O_DIRECT` was rare by default, being enabled by default in only two out of 16 example programs (Firebird and MongoDB); however, 11 out of those 16 programs (Nginx, AOLserver, OpenSSH, Squid, Firebird, Hypertable, MySQL, PostgreSQL, MomgoDB, MariaDB, and SQLite) made use of `open()` with a variable-based flags setting that could be corrupted and thus turn on use of `O_DIRECT`. Based on these results,

the opportunity for the FDMA attack exits in many real-world applications.

6 DISCUSSION AND LESSONS LEARNED

In this paper, we demonstrated how QUASAR can help analyze the extent to which attack capabilities may be constrained in the context of an overall attack method, and allow us to quantitatively assess the coverage provided by any particular set of defenses. QUASAR is well-suited for defense planning and research allocations.

However, we also discovered a number of limitations to the general approach of modeling attack capabilities and defenses for satisfiability testing. The effort needed to model attacks at useful levels of abstraction is high, taking large amounts of time and requiring a high degree of domain expertise. During the process of model construction, we had to walk a fine line between defining overly broad capabilities and too finely-grained capabilities. If the model grows too large, the NP-Hard process of compiling into SDDNNF begins to take longer; simultaneously, a model that is too abstract and lacks sufficient atomicity is unlikely to be constrained by realistic defenses, which would reduce the value of QUASAR as an analysis tool.

The definition of defenses in the knowledge base is a similarly challenging problem, requiring an in-depth understanding of the specific defense, of the existing set of attack capabilities, and the ability to express the constraints that the defense puts on those capabilities. As discussed in Section 3, we defined a domain-specific language to do so. This language is expressive and allowed us to encode most defenses for memory corruption using only a few statements. In theory, a defense could be significantly complicated enough as to make the encoding task difficult in even a specialized language, but we found this to be rare in the case of memory corruption.

These limitations are largely mitigated by the fact that most of the utility of QUASAR comes from its use after the ACG construction has been completed. However, we must also stress that maintenance of attack capabilities and defenses in the knowledge base is important, as the nature of attack classes can change over time as new offensive and defensive paradigms are developed. In many cases, portions of the model may need to have additional layers of atomicity added to fully cover recent developments.

7 LIMITATIONS

QUASAR provides a mechanism for encoding and automatically reasoning about domain knowledge bases. While we find this approach useful for defense planning and research, it does have a number of limitations that makes it unsuitable for solving certain kinds of problem. Formal proofs of security, for example, are impossible using QUASAR for two reasons. First, the soundness of a conclusion (*e.g.,* that an attack strategy actually bypasses a defense) can only be verified empirically. Absent a formal mathematics of cybersecurity which captures the dynamics of all possible defenses and attacks, the most QUASAR can do is provide recommendations for which attack strategies are likely worth testing empirically.

Second, the model is necessarily incomplete. QUASAR formalizes *existing* domain knowledge. It does not, and cannot, provide any formal guarantees about what domain experts do not know. This is analogous to the problem of 'unknown unknowns', and will persist as a problem for as long as the memory corruption domain remains outside the realm of a formal mathematics of cybersecurity. For example, defense coverage can only be assessed against the space of attack strategies known to the domain expert who creates the model. Entirely novel attacks are not considered, and if developed may well bypass a defense. Rowhammer [20] is one such example. It operates outside the normal dynamics of memory corruption, and was not anticipated by our analysis.

Another limitation of QUASAR is that its measures of attack criticality and defense coverage do not cleanly map to the real world. Attack criticality measures how many distinct strategies rely on a shared capability, but not how difficult that capability is to acquire, the probability of it working, the cost of implementation, *etc.* In reality, attackers may prefer to use a set of low-criticality capabilities because they are easier to acquire, and in the end only one successful strategy is needed to compromise a system. The same limitations apply to defense coverage, which measures the number of attack strategies blocked by a set of defenses. In addition to being incomplete with respect to attack strategies, it also does not provide any indication of how 'hard' it is to attack a system in the real world given those strategies which remain.

8 RELATED WORK

The closest related work to QUASAR is Attack/Defense trees (ADTrees) [21], which is also based on modeling attacker strategies as dependency relations rooted in a high-level goal. Defenses in this formalism, however, act as counter-actions to attack actions in a game played between the attacker and defender, rather than constraints over attack capabilities. The game is won when either the attacker or defender succeeds by preventing the other from achieving their top-level goal. This all-or-nothing approach means that ADTrees do not support quantitative analysis such as defense coverage or attack criticality.

Attack Countermeasure Trees [28] are a stochastic approach to modeling attacks and mitigation strategies. They support single- and multi-objective optimization for determining returns on the attacker and defender investments. Defenses cannot arbitrarily constrain attacker options, however, nor can they be easily compared against one another.

Szekeres *et al.* [39] provide a comprehensive semi-formal taxonomy of memory corruption attackers and defenses. This taxonomy is not intended for automated analysis, however, and does not provide a way to quantitatively analyze attacks or defenses.

The work by Skowyra *et al.* [36] is a precursor to QUASAR, which is based on qualitative analysis of attacks and defenses using Boolean SAT solvers. Thus, it does not support measures for comparing defenses or identifying high-impact attacks.

9 CONCLUSION

The framework and methodology as described in this paper and implemented via QUASAR, allows us to quantify the extent to which different attack capabilities are constrained with respect to an overall attack method, determine the coverage of a defense with respect to constraints on attack capabilities, and identify high-impact priority areas for both attackers and defenders. We have done so by

fully specifying a model for the memory corruption domain and have run an analysis on it, including the identification of the top five high-priority areas. This identification was subsequently confirmed by the recent publication of various attacks in four out of five of those areas, and our own proof-of-concept exploit against the fifth. We believe that QUASAR can serve as a tool for defense planning and strategic thought about defensive capabilities in large organizations and in research planning and priority allocations.

Future work includes expanding the knowledge base for additional attack domains (*e.g.*, web-based attacks), defining and computing improved measures of functional defenses comparisons, and the operationalization of the tool for use in different settings.

REFERENCES

[1] CVE-2015-7545. Available from MITRE, CVE-ID CVE-2015-7545. (feb 2016), https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-7547

[2] Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Proc. of ACM CCS. pp. 340–353 (2005)

[3] Anderson, J.P.: Computer security technology planning study. volume 2. Tech. rep., DTIC Document (1972)

[4] Baratloo, A., Singh, N., Tsai, T.K., et al.: Transparent run-time defense against stack-smashing attacks. In: USENIX Annual Technical Conf. pp. 251–262 (2000)

[5] Becher, M., Dornseif, M., Klein, C.N.: Firewire: all your memory are belong to us. Proceedings of CanSecWest (2005)

[6] Biere, A., Heule, M., van Maaren, H.: Handbook of satisfiability, vol. 185. IOS press (2009)

[7] Bigelow, D., Hobson, T., Rudd, R., Streilein, W., Okhravi, H.: Timely rerandomization for mitigating memory disclosures. In: Proc. of ACM CCS (2015)

[8] Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T.R.: Control-flow bending: On the effectiveness of control-flow integrity. In: Proc. USENIX Security (2015)

[9] Carrier, B.D., Grand, J.: A hardware-based memory acquisition procedure for digital investigations. Digital Investigation 1(1), 50–60 (2004)

[10] Cowan, C., Beattie, S., Day, R.F., Pu, C., Wagle, P., Walthinsen, E.: Protecting systems from stack smashing attacks with stackguard. In: Linux Expo (1999)

[11] Crane, S., Liebchen, C., Homescu, A., Davi, L., Larsen, P., Sadeghi, A.R., Brunthaler, S., Franz, M.: Readactor: Practical code randomization resilient to memory disclosure. In: Proc. of IEEE S&P (2015)

[12] Darwiche, A.: Decomposable negation normal form. Journal of the ACM (JACM) 48(4), 608–647 (2001)

[13] Darwiche, A.: On the tractable counting of theory models and its application to truth maintenance and belief revision. Journal of Applied Non-Classical Logics 11(1-2), 11–34 (2001)

[14] Darwiche, A.: A compiler for deterministic, decomposable negation normal form. In: AAAI/IAAI. pp. 627–634 (2002)

[15] Evans, C.: Advancing exploitation: a scriptless 0day exploit against linux desktops (2016), http://scarybeastsecurity.blogspot.com/2016/11/0day-exploit-advancing-exploitation.html

[16] Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H., Sidiroglou-Douskos, S.: Control jujutsu: On the weaknesses of fine-grained control flow integrity. In: Proc. of ACM CCS (2015)

[17] Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.W.: Ilr: Where'd my gadgets go? In: Proc. of IEEE S&P (2012)

[18] Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-oriented programming: On the expressiveness of non-control data attacks. In: Proc. of IEEE S&P (2016)

[19] Kil, C., Jun, J., Bookholt, C., Xu, J., Ning, P.: Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In: Proc. of ACSAC'06 (2006)

[20] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In: ACM SIGARCH Computer Architecture News. vol. 42, pp. 361–372 (2014)

[21] Kordy, B., Mauw, S., Radomirović, S., Schweitzer, P.: Foundations of attack–defense trees. In: International Workshop on Formal Aspects in Security and Trust (2010)

[22] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-pointer integrity. In: OSDI (2014)

[23] Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: Sok: Automated software diversity. In: Proc. of IEEE S&P (2014)

[24] Madan, B.B., Goševa-Popstojanova, K., Vaidyanathan, K., Trivedi, K.S.: A method for modeling and quantifying the security attributes of intrusion tolerant systems. Performance Evaluation 56(1), 167–186 (2004)

[25] Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., Kirda, E.: G-free: defeating return-oriented programming through gadget-less binaries. In: Proc. of ACSAC (2010)

[26] OpenBSD: Openbsd 3.3 (2003), http://www.openbsd.org/33.html

[27] PaX: Pax address space layout randomization (2003), http://pax.grsecurity.net/docs/aslr.txt

[28] Roy, A., Kim, D.S., Trivedi, K.S.: Cyber security analysis using attack countermeasure trees. In: Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research (2010)

[29] Rudd, R., Skowyra, R., Bigelow, D., Dedhia, V., Hobson, T., Crane, S., Liebchen, C., Larsen, P., Davi, L., Franz, M., Sadeghi, A.R., Okhravi, H.: Address-Oblivious Code Reuse: On the Effectiveness of Leakage-Resilient Diversity. In: NDSS (2017)

[30] Sang, F.L., Nicomette, V., Deswarte, Y.: I/o attacks in intel pc-based architectures and countermeasures. In: SysSec Workshop (SysSec), 2011 First (2011)

[31] Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.R., Holz, T.: Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In: Proc. of IEEE S&P (2015)

[32] Seibert, J., Okhravi, H., Soderstrom, E.: Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In: Proc. of ACM CCS (2014)

[33] Sevinsky, R.: Funderbolt: Adventures in thunderbolt dma attacks. Black Hat USA (2013)

[34] Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proc. of ACM CCS. pp. 552–561 (2007)

[35] Sheyner, O., Haines, J., Jha, S., Lippmann, R., Wing, J.M.: Automated generation and analysis of attack graphs. In: Proc. of IEEE S&P (2002)

[36] Skowyra, R., Casteel, K., Okhravi, H., Zeldovich, N., Streilein, W.: Systematic analysis of defenses against return-oriented programming. In: Proc. of RAID (2013)

[37] Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.R.: Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: Proc. of IEEE S&P (2013)

[38] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., Walter, T.: Breaking the memory secrecy assumption. In: Proc. of EuroSec'09. pp. 1–8 (2009)

[39] Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal war in memory. In: Proc. of IEEE S&P (2013)

[40] Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In: Proc. of ACM CCS (2012)

[41] Zou, C.C., Towsley, D., Gong, W.: Modeling and simulation study of the propagation and defense of internet e-mail worms. IEEE TDSC 4(2) (2007)