Vulnerability-Tolerant Secure Architectures

(Invited Paper)

Todd Austin[1], Valeria Bertacco[1], Baris Kasikci[1], Sharad Malik[2], and Mohit Tiwari[3]

[1]University of Michigan
[2]Princeton University
[3]University of Texas at Austin
Corresponding Email: austin@umich.edu

Abstract

Today, secure systems are built by identifying potential vulnerabilities and then adding protections to thwart the associated attacks. Unfortunately, the complexity of today's systems makes it impossible to prove that all attacks are stopped, so clever attackers find a way around even the most carefully designed protections. In this article, we take a sobering look at the state of secure system design, and ask ourselves why the "security arms race" never ends? The answer lies in our inability to develop adequate security verification technologies. We then examine an advanced defensive system in nature -the human immune system -and we discover that it does not remove vulnerabilities, rather it adds offensive measures to protect the body when its vulnerabilities are penetrated. We close the article with brief speculation on how the human immune system could inspire more capable secure system designs.

1. Introduction

It seems that one inescapable truth in computer security is that, regardless of the system, attackers always find vulnerabilities to exploit, and if those vulnerabilities are fixed, they will soon find ways around those protections. 1bis "security arms race," as it is called, persists because, despite advances in security verification [1, 2], the complexity ofreal systems precludes the possibility of proving that a design cannot be attacked.

1.1. Why the Security Arms Race Never Ends

In an ideal world, hardware and software designers could harness powerful security analysis tools that would implement proofs for each vulnerability of concern, in an effort to show that for a particular system:

$$\forall \textit{(programs, inputs, vulnerabilities)} \ \textit{(system is secure)}$$

Unfortunately, the creation of truly secure systems is not possible for arbitrary systems running arbitrary software, because:

- Itis not possible to easily define what it means for the system to be secure. Specific properties such as confidentiality and integrity are useful, but are incomplete notions of security.
- Even for a specific vulnerability and definition of system security, the resulting verification problems are intractable and tend not to scale to practical systems.
- It is not possible to know all possible threats and vulnerabilities a priori. This lack of knowledge is exactly what attackers exploit.

While much progress has been made in formal verification, success is limited in type and scale of the systems addressed (*e.g.,* SLAM [3]) or scope of the analysis (*e.g.,* Whoop [4]). Thus given the complexity of today's systems, the goal of securing general software and hardware remains unreachable through security proofs.

The incompleteness and intractability of formalized security verification techniques is further exacerhated by the fast-growing complexity of modern systems. Fueled by the computational and memory resources made available by Moore's Law, designers are continually building systems with increased complexity. A recent study of lines of code for avionic systems found that the code size in Boeing and Airbus commercial jets is growing at a rate of 2x every four years [5]. Another study for Ford vehicles found a !Ox increase in the size of on-board software over a 3 year period [6]. Similar trends can be seen in hardware design complexity. A recent study of Apple SoCs found that the number of accelerators grew from 9 to 37 in less than decade, with no sign of slowing [7].

1.2. Functional Verification Isn't Scaling to the Challenge

Functional verification works to find system design errors, many of which are vulnerabilities that could be exploited to penetrate security. During functional testing, which is the workhorse of verification technologies, engineers use hand-made tests, random testing, and formal analysis to exercise as much of the functionality of the design as possible, given engineering resources [8]. Again, due to the complexity of modern systems, there is no hope to fully test the system, so verification engineers direct tests toward likely runtime states.

For functional verification, it is important to test the most likely states to be visited while the system is in operation. Thus, "coverage metrics" are used to assess to what extent these likely states have been exantined [9]. Studies have shown that while a design possesses a multitude of reachable states, normal operation only touches a tiny fraction of these states [10, 11], leading to verification of likely states being a very effective functional verification approach.

While "best effort" serves functional verification well, it

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted To copy othe.rwi.se, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions frompermissions@acm.org.
ICCAD '18, November 5--8, 2018. San Diego, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5950-4/1 8111... $15.00
https://doi.org/10.1145/3240765.3273057

falls way short of what is necessary for high-quality security verification. Functional verification wants to find the bugs that are likely to be encountered with real-world software, thus, it is possible to find most of them by running real-world software on test systems. Security verification, on the other hand, is a very different endeavor. Attackers do not seek out likely execution scenarios, as these are quite unlikely to have vulnerabilities. Instead, attackers seek out the most unlikely configurations, because these have not been well tested. For example, if an attacker wants to exploit a program asking which day of the month you were born, their inclination is to enter a negative number, to see how the program responds to the unexpected. Much research supports this observation that attackers seek unlikely execution scenarios [12, 13]. The implication for verification engineers is that testing the most likely execution scenarios provides very little value inhardening a system against security attacks. Instead, security verification engineers have to explore the most unlikely execution scenarios, and if they want high coverage vulnerability coverage, they must work toward verifying all possible system configurations – which of course is impractical for non-trivial systems. As such, verification methods, which already fall short for functional verification, fall even shorter for security verification. In the end, large complex designs are released to the public with yet-to-be-found security vulnerabilities lying within them.

Given the higher bar for security verification, two additional forms of verification have emerged: red teaming and bug bounties. Red teaming engages a team of attackers to ethically attack a system to discover its vulnerabilities [14]. 'fypi.cally, the team is hired externally from a white-hat attacker company, although some larger organizations have in-house red teams. Red teams employ the same tools that black-hat attackers use to penetrate the system, ranging from prepackaged exploits, to random test-generation tools, and hand-crafted attacks. If the red team is very skilled, the approach can be unmatched for the degree and quality of the vulnerabilities found, but ultimately the system will still possess vulnerabilities.

In a similar vein, the white-hat attack research community, composed primarily of acadentics, security research companies, and independent researchers, expends significant effort to attack real systems to expose vulnerabilities before they can be exploited. The white-hat attack community is essentially a publicly funded redteam, providing open access to the discovered vulnerabilities. Support for white-hat attacking research is a powerful measure to find important threats. Ifwell funded, the white-hat attack community can help designers understand the threat landscape, so as to better craft future secure systems and their protections. In the US, the National Science Foundation (NSF) has long supported white-hat attack research through the SaTC program [15], although supported efforts typically include the development of protections as well.

Perhaps the most effective means to find security vulnerabilities that are ntissed by security verification is to purchase them from would-be attackers before they are used to harm users, using what are called "bug bounties" [16]. Bug bounties are paid to attackers that find a yet-unknown security vulnerability. If the vulnerability is responsibly disclosed to the holder of the bounty and verified, the attacker will typically be paid a bounty in proportion to the vulnerability'• severity. Severity is often scored using the industry standard Common Vulnerability Scoring System (CVSS) [17]. Bug bounties have been quickly growing in number and amounts in the last few years because it has been shown to be an effective method to entice attackers to disclose a uew vulnerability, rather than use it to harm the users of a vulnerable system. For example, Intel instituted its security bug bounty program in March 2017, and expanded its payouts to up to $250,000 in Fehruary 2018 [18]. Yet, even with lucrative bug bounties, attackers will often choose to retain vulnerabilities for their own use.

2. InSearch of Advanced Defense Systems -The Human Immune System

To move heyond the lintitations of security verification, in this work we explore *vulnerability-tolerant secure systems.* These systems, despite having exposed vulnerabilities, are still resistant to attack because they incorporate defenses that make them impractically difficult to penetrate. Surprisingly, there is a prime example of a "security" system of this form in nature in the human immune system. Consequently, we have studied the human immune system for inspiration on how to achieve the goal of a vulnerability-tolerant secure system.

2.1. Properties of the Hnman Immune System

The human immune system is a very effective analogy for system security, where the "system"is the human body, the "security defenses" are the immunity mechanisms inthe body, and the "attackers" are all forms of disease. It is very instrnctive to examine the human immune system, since its capabilities are significantly more advanced that even today's most advanced security technologies. Specifically, the human immune system has the following highly desirable defense properties:

Vulnerability Tolerant – The human immune system is not focused on finding and fixing vulnerabilities, in fact, our bodies are constantly under attack from diseases in our environment. Instead, the human immune system focuses on thwarting the attacks of diseases once they enter the body.

Antifragile – Our immune system, when exposed to a pathogen with a successful recovery, gains strengthened protections against that pathogen for years into the future. This very powerful property of the immune systems forms the basis for vaccinations, where the immune system is exposed to weak strains of a disease to build up defenses that can stop later exposures to the same disease.

Reinforceable – When we feel sick, our body engages additional defense mechanisms to strengthen defenses. For example, when the body experiences an infection, a fever is induced to slow the growth of the invading pathogens. Similarly, our ability to learn and invent has further extended this reinforcement capability -when we feel prolonged sickness or discomfort, we can go to the doctor for treatment.

2.2. How Does the Immune System Work?

It is interesting to study the details of the immune system, to better understand how it gains its powerful defensive prop-

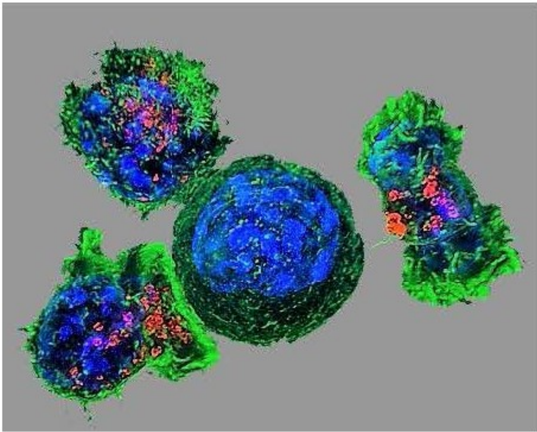

Figure 1: T Cells Attacking a Cancer Cell. In the figure. three T-cells are surrounding and attacking a cancer cell (center). T-cells represent a primary component of the immune syst.em's vulnerability tolerant defense capability. Used with permission from [19].

erties. The vulnerability tolerance of the immune system is attained through an active set of measures to seek out and destroy pathogens that have entered the body. Of particular note in this regard are the white blood cells and their T-cell variant T-cells form a family of cells that attack pathogens, as shown in Figure 1. The effector T-rells, which attack invading pathogens, have a special capability to differentiate between host cells and invading disease.

The binding mechanisms inthe T-cells can be adapted over time to better attack commonly occurring pathogens through a process called somatic hypennuta.ti.on, which rearranges the DNA of T-cells to induce diversity in their properties. This process ensures that T-rells exist at all times in many variants; thus, if a pathogen were to attack a T-cell variant, there would be many other variants that could still protect the body.

The somatic hypermutation process is an example of a moving target defense. A *moving target defense* is a defensive system that deliberately introduces entropy in the system to increase uncertainty [20]. By introducing uncertainty, the attacker must probe the system to identify uncertain values. In the immune syst.em, this would require a pathogen, such as a virus, to adapt to the a specific T-cell variant. However, most T-cells only live a few months, so they are a quickly moving target, such that most infections cannot adapt fast enough to overtake the body's defenses.

The anti.fragility of the immune system is also an interesting feature to consider. When T-cells identify a pathogen to attack, they split into an effector T-cell and a memory T-cell. The effector T-cell attacks the pathogen, which is then removed by the spleen. The memory T-cell has an extended lifespan of about 5-10 years that is specifically conditioned to recognhe the specific pathogen in the future and initiate the production of associated effector T-cells. It is the memory T-cells that provide the immune system with its antifragility property.

The reinforcement property of the immune system arises from both innate :responses, such as vomiting or fever, as well as learned responses. Our ability to learn and invent has significantly extended this capability. When we feel sick, we can use over-the-counter medicines or see a doctor for treatment

```
voJ.d target() l
  printf("You overflowed successfully, gg"l;
  exit(0l;
J
voj,d vulnerable(ahllr•strl) {
  abar buf[SJ;
  strcpy(buf, strl l;
}
int main() <
  vulnerable("ffffffffffffffff\sf0\.01\xOl\•OO•);
  printf(wThiB only prints in normal control flow•);
```

Figure 2.: Example Attack Code. This code performs a buffer overflow attack in the call to *nrr:py().* The address of *target()* is oxooo1olf o, which is expressed in the overflow string as "\xflJ\xOl\xO1\x.00". We note that the attack code utilizes unspecified semantics associated with array overflows, stack frame organization, and code addresses.

In addition, this capability allows us to better maintain good health, via eKerCise, better diet, and smarter life choices. This mechanism is incredibly powerful, having doubled average human lifespan from 35 to 70 years in the last 500 years.

Clearly, when one considers the human immune system, it is significantly more advanced than any comparable computer or network security system. In fact, if the human immune system were equivalent to today's secure system designs, we would all be living our lives in hazmat suits, since our primary defense against disease would be to ensure that no pathogen ever entered our body l While it is easy to see that immune-system inspired defenses benefit human populations, a carefully designed defense can protect individual computer systems as well. We outline such a system in the next section.

3. Crafting a Vulnerability-Tolerant System

While we see that a laudable long-term goal is to design a secure system that has all the desirable properties of the immune system, at this stage of our efforts, we are focusing on the property of vulnerability tolerance. *As* our efforts continue, we will expand our focus to antifragility and later reinforcability. Consequently, to build a secure system that is vulnerability tolerant, it is necessary to construct a computing platform that is benign to regular programs but hostile to malicious programs.

3.1. Regolar Programs are from Mars; Malicious Programs are from Venus

To make life easy for regular programs but hostile for malicious programs, we need to carefully consider the differences between these two types of programs. Figure 2 shows malicious code that performs a buffer overflow attack in *strcpy().* At first glance, one might limit the difference between the malicious and non-malicious code to the observation that the malicious code overflows variable *buf[]* to overwrite the return address of *vulnerable( ),* and the non-malicious version (when the string passed to *strcpy()* is length 4 or less) does not However, if one takes a more nuanced view, more distinctions exist *We olnene that regu'larprognuns utilke rpecified program-level semantics, while malicious programs lean heavily on unspecified execution-le1 1el snnantics.*

Unspecified semantics are execution-level semantics that are not explicitly documented, typically because they are properties of the underlying implementation. Examples of undefined

semantics include: out-of-bound array access semantics, uninitialized variable values, execution tinting, microarchitectural sharing characteristics, *etc.* While savvy programmers know that these values exist and often are static, they would never build a program that relied on these unspecified semantics, as these properties could easily change from one build of the program to the next, or from one CPU architecture to another.

Malicious programs, in contrast, routinely utilize a variety of undefined semantics to subvert security measures. For example, the code in Figure 2 copies memory past the end of the array *buff]* (unspecified semantic: out-of-bound array access), which overwrites the return address (unspecified semantic: return address location) with the value of 0x0 00 l0 lf 0 (unspecified semantic: code address). We performed an extensive study of attacks across a wide range of attack classes and we found, without counterexample, that malicious programs extensively utilize unspecified semantics. We are not the first to observe this as debugging tools have long focused on finding unspecified semantics *(e.g.,* UBSan [21]). Indeed, programs can utilize unspecified semantics without malice, as we have observed in a few isolated cases in code we have analyzed.

3.2 Boosting Uncertainty with Moving Target Defenses

Given that malicious programs utilize unspecified semantics and regular programs avoid them, an attractive approach to vulnerability-tolerant secure design is to keep well-defined semantics as such and randomize the unspecified program semantics each time a program is executed. The use of randomization **on unspecified semantics as a defensive measure** is an example of a *moving target defense* [20]. Moving target defenses deliberately introduce change into a system, which in turn increases the attacker's uncertainty of the system's state.

By randomizing the undefined semantics of a program each time it runs, the attacks built for a system with static unspecified semantics will no longer work, as key attack values have become randomized. *Moving target defenses force the attacker to probe the system for the unkntJWn values needed to perpetrate an attack.* Probing involves conducting an experiment where the outcome yields some information about the uncertain value. For example, if the code in Figure 2 must contend with a moving target defense applied to the location of the code, the attacker must first probe the system to determine the address of *target().* The malicious program could (naively) introduce a memory scanner that searches the 48-bit RISC-V address space for the function *target().* With extreme patience, the scanner will eventually locate *target()* and then the corrected string can be injected into the call to *strcpy().*

Fortunately, we have a prime example of how the attacker community responds to moving target defenses in address space layout randomization (ASLR) [22]. ASLR randomizes the location of the code, heap, and stack each time a program runs, which bas the effect of applying a moving target defense to code/data locations and pointer values. Today, ASLR is widely deployed; Table I lists ASLR defenses and the subsequent probes attackers developed to overcome the resulting uncertainty. The table also shows the entropy of defenses (or number of bits recovered) and estimated time for an at-

Type	Name	Entropy / Bits Recovered	Tllle (s)
Defense	32-bit PaX ASLR [22]	16 bits	n/a
- Attack	Blind calls (Pfafi) [24]	16 bits	216
Defense	64-bit ASLR	30 bits	nia
- Attack	BROP [25]	30 bits	1,200
- Attack	CROP [26]	30 bits	14,580
- Attack	Dedup Est Macbina [27]	30 bits	1,800
- Attack	JUMP over ASLR [28]	9 bits	0.06
- Attack	AnC [23]	30 bits	150

Table 1: ASLR Defenses and Probing Attacks. This time-ordered table lists ASLR defenses and attack probes used to recover randomized addresses for that defertse. It is evident that strong moving target defenses (e.g. 64-bit ASLR) result in long probe times.

tack. While one-time moving target defenses like ASLR are very powerful stumbling blocks for attackers, today they form only temporary barriers since, with enough time and ingenuity, attackers can overcome all types of ASLR. For example, the AnC attack is able to recover a full 48-bit x86_64 virtual address in about 150 seconds [23]. As shown in Table 1, attacking moving target defenses with high entropy requires either *i)* more advanced probing techniques and/or *ii)* significant probe times. Given this trend, we would expect increased probe time for systems with advanced moving target defenses.

3.3 Ensembles of Moving Target Defenses with Chum

Moving target defenses that randomize program state each time the program is run (like ASLR) are powerful defenses, but ultimately, with enough resources the attacker can succeed in probing the system for the information needed to attack. If we want to bring moving target defenses to the next level, we must leverage *ensembles of moving target defenses (EMTDs)* to boost uncertainty and combine them with a runtime re-randomization capability, which we call *churn.*

The introduction of a churn mechanism can thwart even the most patient attackers. Churn re-randomizes the values that attackers need to craft successful attacks. Thus, *with the high levels of uncertainty provided by ensembles of moving target defenses andfast churn rates, attack probes will fail to discern the unkntJWn values for which they search.* For example, an attack probe for the code in Figure 2 searches the address space for the function *target().* With churn, the function *target()* could repeatedly move in the address space as the search progresses, thus, the probe would only succeed on the infinitesimal chance that *target()* moves to the immediate vicinity of the attack probe's search.

To make EMTDs with churn an even more hostile environment for attackers, it is possible to combine churn with an *attack detector.* The detector can watch for operations indicative of undefined semantics *(e.g.,* arithmetic on code pointers) and then immediately initiate a churn cycle in response to a possible attack. Under normal operation, churn could proceed at a rate that has negligible performance impact but is orders-of-magnitude faster than expected probe times. When the detector is triggered, a churn cycle is immediately started, which severely limits the time available for a successful attack probe. In our analyses, we have observed that regular

programs rarely trigger the detector while attack probes are rife with operations that trigger the detector, such that attacks would indeed be forced to contend with continuous churn.

To get a sense of how EMTDs with churn could thwart attacks, again consider the buffer overflow attack shown in Figure 2. The buffer overflow attack invokes *target()* when *vulnerable()* returns. The program is malicious because the string passed to *strcpy()* is too long for the array *buff],* allowing the string to overwrite the return address of *vulnerable()* with the entry-point of *target().* With aggressive EM'ID defenses, two complications could be thrown at the attacker: *i)* the address of *target()* could be randomized by underlying defenses, and *ii)* the representation of code pointers (which is expressed in the string as "\xf O \xOl \xOl \xO O") could adopt a randomized representation, such as being encrypted, making the pointer in the string incorrectly encoded as plaintext. To overcome these challenges, the attacker would have to significantly probe the EM'ID-protected system to discover *i)* the location of *target( )* and *ii)* the representation of code pointers. It is quite likely that these probes would take several minutes or more, even with the most advanced probes. Unfortunately (for the attacker), the churn mechanism will re-randomize the address of *target()* and its pointer representation within a short period *(e.g.,* 50 ms). Even worse – if the attacker's probes trigger the attack detector, a new churn cycle will start immediately, which will effectively destroy the address of *target( )* and again re-randomize the representation of code pointers.

Revisiting the question of how vulnerability-tolerant architectures can protect individual systems, we observe that the key theme behind vulnerability-tolerant defenses is to layer several independent protections such that each prevents an attack from abusing undefined seniantics. Each layer is designed to randomize undefined semantics and thwart attacks and probes that are designed to reverse-engineer them. As a result, while EMTDs are inspired by immune systems in nature, they can be tuned to protect individual systems or even individual classes of attacks.

4. Conclusions

Traditional security protections often fall short because the complexity of proving that protections cannot be circumvented nearly always outstrips the capabilities of existing verification technologies. Inthis article, we speculate on the possibility of vulnerability-tolerant secure system design approaches, inspired in part by the human immune system. Ensembles of moving target defenses with churn bring together multiple defenses to protect a system from attacks. The churning mechanism is able to re-randomize these values at runtime to stop attempts to de-randomize the system, without inlpacting regular programs. Together, these protections have great potential to advance the state of the art in secure system design.

5. Acknowledgments

This work was supported by DARPA under Contract HROOl l-18-C-0019. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[1] C. Hawblit7.el, J. Howell, M. Kapritsos, J. Lorch, B. Pamo, M. L. Roberts, S. Setty, and B. Zill, "lronfleet: Proving practical distributed systems correct:' in *SOSP'l5,* October 2015.

[2] B. Born!, C. Hawblit7.el, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Pamo, A Rane, S. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *USENIX Sec'l7,* 2017.

[3] T. Ball and S. K. Rajamani, "The SLAM project Debugging system software via static analysis," *SIGPLAN Not.,* vol. 37, Jan. 2002.

[4] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric, "Fast and precise symbolic analysis of concurrency bugs in device drivers," in *30th IEEE/ACM lnt'l Conj. on Automated Software Engineering,* 2015.

[5] J. Hansson, S. Helton, and P. Feiler, "ROI analysis of the system architecture virtual integration initiative," Tech. Rep. CMU/SEI-2018-TR.-002, Carnegie Mellon University, 2018.

[6] R. Saracco, "Guess what requires 150 million lines of code...." IEEE Future Directions Tech Blog, 2016. http://sites.ieee.org/futuredirections/201 6/01/13/guess-what-requires-150-million-lines-of-code/.

[7] Y.S. Shao, B. Reagen, G. Wei, and D. Brooks, "The Aladdin approsch to accelerator design and modeling," *IEEE Micro,* vol 35, May 2015.

[8] B. Bentley, "Validating a modern microprocessor," *Computer Aided Verification,* 2005.

[9] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design and Test,* vol. 18, July 2001.

[10] I. Wagner and V. Bertacco, "Engineering trust with semantic guardians," in *2007 Design. Automation Test in Europe,* April 2007.

[11] E. Schnarr and J. R. Larus, "Fast out-of-order processor simulation using mem.oization," in *lnt'l Conj. on Architectural Supportfor Programming Languages and Operating Systems, ASPW S* VIII, 1998.

[12] W. Arthur, B. Mammo, R. Rodriguez, T. Austin, and V. Bertacco, "Schnauzer: Scalable profiling for likely security bug sites," in *lnt'l Symp. on Code Generation and Optimization (CGO),* Feb 2013.

[13] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing." *Queue,* vol 10, pp. 20:20-20:27, Jan. 2012.

[14] M. Fenton, "Restoring executive confidence: Red team operations," *Network Security,* Nov 2005.

[15] "Secure and trustworthy cyberspace (SaTC)." https://www.nsf . gov/f unding/pgm_summ. jsp?pims_id=504709.

[16] H. Hata, M. Guo, and M. A Bahar, "Understanding the heterogeneity of contributors in bug bounty programs," in *Int'l Symp. on Empirical Software Engineering and Measurement,* ESEM '17, 2017.

[17] "Vulnerability metrics." https: //nvd.nist .gov/vuln-metrics.

[18] "Expanding Intel's bug bounty program: New side channel prcr gram. increased awards." https: //newsroom. intel .com/news/ expanding-intels-bug-bounty-prograrn/, Feb.201 8.

[19] "T cell." https: //en. wikipedia. org/wiki/T_cell.

[20] S. Jajoclia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, *Moving Target Defense: Creating Asymmetric Uncertaintyfor Cyber Threats.* Springer Publishing, 2011.

[21] Apple Corp., "Undefined Behavior Sanitizer." https: //developer.apple.com/documentation/code_diagnostics/ undef ined_behavior_sanitizer.

[22] PaX Team, "PaX address space layout nmdomization (ASLR)." http: //pax .grsecurity.net/docs/aslr.txt, 2003.

[23] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU," in *NDSS,* Feb. 2017.

[24] H. Shacham, M. Page, B. Pfaff, E.-1. Goh, N. Modadugu, and D. Boneh, "On the Effectiveness of Address-space Randomization," in *Conj. on Computer and Communications Security,* CCS '04, 2004.

[25] A. Bittau, A. Belay, A. Mashtizadeh, D. Maziilres, and D. Boneb, "Hacking Blind," in *IEEE Symp. on Securily and Privacy,* SP '14, 2014.

[26] R. Gawlik, B. Kollenda, P. Koppe, B. Gannany, and T. Holz, "Enabling client-side crash-resistance to overcame diversification and information hiding," in *NDSS,* 2016.

[27] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup est machina: Memory deduplication as an advanced exploitation vector," in *IEEE Symp. on Security and Privacy (SP),* May 2016.

[28] D. Evtyushkin, D. Ponomarev, and N. AOO-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *lnt'l Symp. on Microan:hitecture (MICRO),* Oct 2016.