From Dependability to Security—A Path in the Trustworthy Computing Research

Shuo Chen

Abstract The societal importance of trustworthy computing has become more and more obvious. It has two distinguishable yet related aspects: dependability and security. In this chapter, I will explain the commonality and difference of the two, and use my own experience as an example to show how a researcher grows his/her expertise through the dependability research and the security research.

1 About Trustworthiness

A fundamental question in computing is how to establish trustworthiness of computational results produced by a real-world system. When discussing the concept of trustworthiness, we must consider the adversary model. The adversary can be phenomena in the nature (e.g., hardware transient errors, communication disruptions and human errors) or intentional human attackers. The former is often considered as the adversary model for *dependability*, and the latter is for *security*. From the perspective of the system designers, implementers and operators, trustworthiness means that the system should be able to withstand these adversaries.

Although the two adversary models are distinguishable, the insights from dependability research and security research are coherent. For example, bit-flip is a basic adversary model, originally in the context of dependability. However, people's understanding about bit-flip has been evolving over a long time. It is now a topic frequently studied in the security community. In addition to the adversary model, formal verification and distributed consensus are also topics evolving from dependability to security. In the rest of this chapter, I provide my perspective in these areas.

S. Chen (✉)
Microsoft Research Asia, Beijing, China
e-mail: shuochen@microsoft.com

L. Wang et al. (eds.), *System Dependability and Analytics*, Springer Series in Reliability Engineering, https://doi.org/10.1007/978-3-031-02063-6_4

2 The Evolution of the Bit-Flip Adversary Model

A useful methodology to evaluate system dependability is fault injection. A fault injector simulates faults that the target system may encounter when it is deployed in the real world. Fault injectors often implement the bit-flip functionality. The functionality targets registers, data memory or code memory. During a test execution of a program, a bit is chosen to be flipped based on a pre-determined distribution. The execution is then resumed. There are two scenarios with high probabilities. The first is that the execution finishes with a correct result. This scenario is often referred to as "fault not manifested". The second high probability scenario is when the execution results in a crash or other exceptions. This is often referred to as "fail silence". Usually, "fault not manifested" and "fail silence" are considered the expected outcome without serious bad consequences. However, there is a non-negligible probability that the execution finishes but produces an incorrect result. This is often referred to as "fail-silence violation". It is the most interesting scenario for investigation.

2.1 Security Consequences Caused by Bit-Flips

My initial knowledge about the security consequences of random bit-flip faults comes from Boneh et al.'s paper in Eurocrypt'97 [1]. The paper shows that several cryptographic systems will be broken if bit-flips can be intentionally introduced during certain phases of the cryptographic computations. For example, an implementation of RSA is based on the Chinese Remainder Theorem (CRT). Boneh et al. show that if the attacker can introduce a bit-flip fault to cause the RSA algorithm to produce an erroneous signature of a message, and repeat the algorithm without the fault to produce the correct signature of the same message, then the secret signing key will be recovered.

My first two papers, published in 2001 and 2002, investigated the security consequences of bit-flips target Internet server programs (e.g., FTP and SSH) [2] and firewall programs (e.g., IPChains and Netfiler) [3]. My co-authors and I conducted fault injection experiments to show the existence of non-negligible probabilities of fail-silence violations resulting in security consequences. For example, injected faults could cause firewall programs to skip packet-filtering rules, or cause FTP's authentication to be bypassed. While these consequences are not surprising, the fact that the probabilities are non-negligible is.

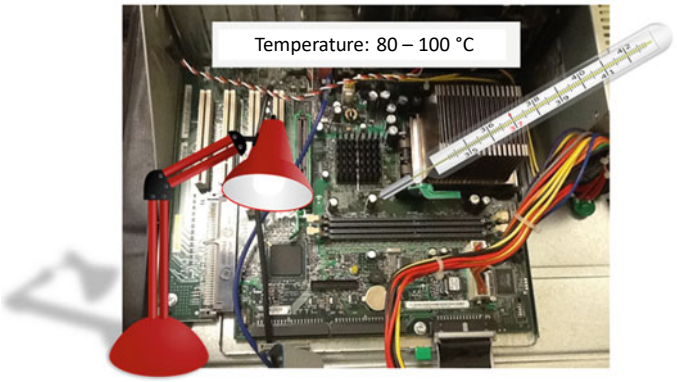

Fig. 1 Using a spotlight to introduce memory errors to JVM

2.2 *Fault Injection as a Weapon*

The papers mentioned above investigate the security consequences with the presence of injected faults, but do not discuss how the faults can be injected in the real-world settings. In this sense, the weaknesses discussed in these papers are not end-to-end exploitable security vulnerabilities.

Our work motivated Govindavajhala et al. to conduct a really surprising experiment in 2003 to show that memory faults can be intentionally injected by heating the PC with a spotlight in a close proximity [4]. (Note that the cover of the PC is removed, so the spotlight is more effective in raising the temperature of the PC's components.) The experiment shows that when the temperature is in the range of 80–100 °C, isolated and intermittent memory errors occur. The authors use this effect to target Java VM (JVM), of which the security assurance crucially depends on type safety. With the presence of memory errors, type safety no longer holds. This means that when the attacker's Java program runs on the JVM, the attacker can take control of the JVM, thus execute arbitrary native code on the victim machine. It is estimated that a single-bit-flip can give a 70% probability for the attack to succeed (Fig. 1).

2.3 *Software Memory Bugs as a Weapon*

The aforementioned research studies give an important insight for a more comprehensive understanding about memory bugs. Before 2005, the attacks exploiting software memory bugs, such as stack overflow, format string vulnerability and heap corruption, focus on the control flow: they use the bugs to rewrite important data that determine the victim program's control flow, e.g., return addresses and function pointers, so that the control flow jumps to an arbitrary binary code supplied by the attacker. They

Table 1 Source code of `getdatasock()`

```
FILE * getdatasock( … ) {
...
seteuid(0);
setsockopt( ... );
...
seteuid(pw->pw_uid);
...
}
```

are referred to as the *control-data attacks*. In response to this attack pattern, many defensive techniques are proposed against them in the research community. Some protect return addresses, such as StackGuard [5] and Libsafe [6]; some rely on control flow integrity for security, such as system call based intrusion detection techniques [7–13], control data protection techniques [14–16], and enforcement mechanisms for non-executable memory [17, 18].

Despite the research community's familiarity of the control-data attack, it is reasonable to ask whether the dominance of control-data attacks is due to an attacker's inability to construct non-control-data attacks, i.e., attacks that do not alter any control-data but still cause security consequences as serious as the control-data attacks. My co-authors and I understood from our previous research that, given a real-world program, its built-in code logic is already susceptible under the bit-flip adversary. In other words, we understood that even if the victim program's control flow is intact, when the code runs on the data slightly corrupted, the consequence can be devastating. This insight might be natural to the dependability community, but was fairly surprising in the security community.

In 2005, we published a paper with the title "Non-Control-Data Attacks Are Realistic Threats" [19]. The paper shows that many types of data, other than control-data, are also crucial to security, including configuration data, user input, user identity data and decision-making data. For example, Table 1 shows the source code of a function in WU-FTPD, which is one of the most widely used FTP servers. WU-FTPD has a format string vulnerability that can be triggered when receiving a "Site Exec" command. Like most other format string vulnerabilities, this vulnerability allows the attacker to overwrite the value of an arbitrary memory location specified by the attacker. Essentially, this vulnerability is a memory fault injector. The function in Table 1 is named `getdatasock`. What it does is to temporarily set the effective UID of the process to the root UID. This is fulfilled by calling `seteuid(0)`. Then, the code does certain operations with the root privilege, such as calling `setsockopt`. In the end, the code restores the effective UID of the process to the user's UID, which is stored in `pw->pw_uid` on the heap. Now, consider what can happen when a format string vulnerability exists. The attacker can exploit the vulnerability to overwrite `pw->pw_uid` to 0, then call function `getdatasock`. The consequence is that `seteuid(pw->pw_uid)` does not restore the process's effective UID, so the attack stays at the root privilege level. All files in the filesystem can be overwritten, including the crucial ones for user authentication, such as /etc/*passwd*

Table 2 Code of `servconnection()`

```
int        serveconnection(int
sockfd) {
char  *ptr;//pointer  to  the
URL.
// ESI is allocated
// to this variable.
...
1: if (strstr(ptr,"/..”))
            reject the request;
2: log(...);
3: if (strstr(ptr,"cgi-bin”))
4:          Handle CGI request
...
}
```

and /etc/*shadow*. This means that the attack obtains the total control of the victim machine.

Another example is about the buffer overflow vulnerability in an HTTP server called GHTTPD. Buffer overflow is also like a memory fault injector, which can overwrite data on the stack. The vulnerable code is shown in Table 2. Function `servconnection()` calls another function `log()`, which contains a buffer overflow bug. A pointer variable `ptr` is on the stack, so it can be overwritten by the attacker because of the bug. Now the question is how the attacker can take control of the victim machine. Although the source code of `servconnection()` is very long, we show two important states in line 1 and line 3. Line 1 rejects any URL that contains a "`/..`" substring. Line 3 implements the CGI functionality of the HTTP server, which allows an HTTP request to invoke an executable on the server. For the security reason, all the invocable executables are stored in a specific path, e.g. `/usr/local /ghttpd/cgi-bin`. An HTTP request http://foo.com/cgi-bin/bar will invoke the executable `/usr/local/ghttpd/cgi-bin/bar`. The checking in line 1 is crucial for the CGI functionality. Suppose a request http://foo.com/cgi-bin/../../../../bin/sh is not rejected, the executable */bin/sh* will be executed, giving the user a command shell. The attacker hence gets the same privilege as the HTTP server. To carry out the attack, the attacker sends a long HTTP request in which the first part is to exploit the buffer overflow bug in order to overwrite the value of `ptr` to be the address of the second part of the request. The second part is the string containing "*/../../../../bin/sh*". This accomplishes the attack.

Our paper [19] investigates other memory bugs in real-world programs. It gives a substantial amount of evidence to show that, when a memory bug allows data corruption, the victim program's existing semantics are usually sufficient to let the attacker get a total control.

2.4 *Rowhammer—A Bit-Flip Security Threat in DRAM*

Since 2014, the security threat caused by bit-flips in DRAM has become an extensively researched topic. The threat and the corresponding exploits are referred to as Rowhammer. The root cause of Rowhammer is the scaling-down of the DRAM process technology. DRAM cells become increasingly likely to charge and discharge between each other, thus have a non-negligible probability to result in bit-flips. This phenomenon was initially described by several patent disclosures by Intel, then studied by the research community. Authors of reference [20] study specifically DDR SDRAM. Figure 2 illustrate the rows of memory cells in the DRAM. One of the rows is the victim row that the attack wants to introduce bit-flips into. It is sandwiched between two adjacent rows. The study demonstrates that the bit-flip probability of the victim row can be substantially increased if the attacker frequently activates the two adjacent rows. Therefore, suppose the attacker can know sufficiently the data contents in these three rows, purposeful bit-flips can be introduced.

There are several follow-up studies based on reference [20]. For example, people understand that ECC (error correcting code) is a technique to mitigate bit-flips, so it is natural to ask whether the Rowhammer threat exists in ECC-protected DRAM. Cojocar et al. conduct a study to reverse-engineer the ECC mechanism. They construct a new Rowhammer attack which can succeed in certain ECC-protected DRAMs [21]. In the separate study, Cojocar et al. develop a methodology to evaluate how cloud servers are vulnerable to the Rowhammer threat [22].

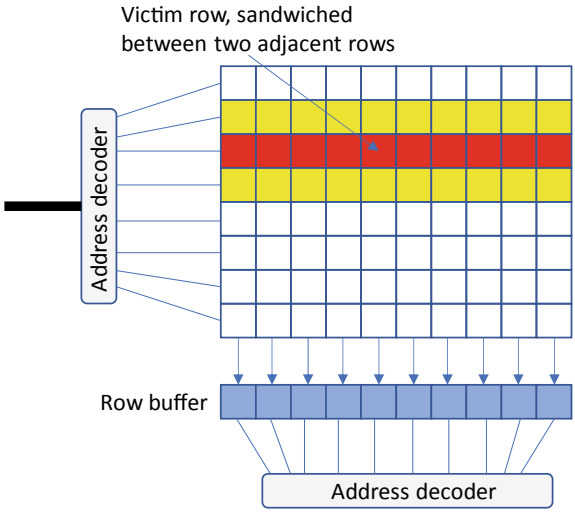

Fig. 2 An illustration about the Rowhammer attack

3 Formal Methods

Formal methods are an important rigorous approach to enhance correctness of a system. My initial knowledge about formal methods was from the reliability context. For example, Rosu et al. developed a formal approach to check the measurement unit (e.g., imperial vs. metric) safety policies for mission-critical programs, such as those written by NASA JPL (Jet Propulsion Laboratory) [23]. This type of safety violations (e.g., an imperial quantity is added to a metric quantity) can hide deeply in a complex program developed by many teams. It is impossible to exhaustively test all the execution paths of the program. Formal methods provide a unique power to statically examine the program to expose bugs with a level of completeness with respect to a given abstraction.

A good example of formal methods for software reliability is the Static Driver Verifier (a.k.a. the SLAM technology) for Windows [24]. Windows needs to accommodate a huge number of device drivers, which run in the kernel space. In the past, Microsoft did not have a quality control mechanism to ensure that the drivers were reliable, so the kernel panic (a.k.a. "blue screen") frequently occurred. Static Driver Verifier enabled Microsoft to implement a Windows driver certification program—a driver succeeding in the verification would be digitally signed by Microsoft, and users were strongly discouraged to install unsigned drivers.

3.1 Formal Methods for Browser Security

Fascinated by the rigorousness of formal methods, I have worked on several security projects that applied formal methods to real-world systems. The first project was about examining Internet Explorer (IE) browser's graphic interface (GUI) logic for security bugs [25]. We studied the GUI code and built a formal model to describe how a user (potentially an attacker) could use Javascript and HTML to spoof the browser's address bar and the status bar. For example, Fig. 3 shows the consequences of two bugs that we discovered. The status bar spoofing bug allowed the attacker to construct a hyperlink in an email. When the user examined the target URL of the hyperlink, the status bar showed https://www.paypal.com, the actual target was the attacker's website. The address bar spoofing bug allowed the attacker to construct a page which could make the address bar and the content window out of sync, so that the address bar (including the SSL certificate) showed https://www.paypal.com, but the content window displayed the attacker's website. Obviously, the combination of status bar spoofing and address bar spoofing would make a powerful phishing attack.

In this work, we used the Maude rewriting logic system [26] to model IE's GUI logic, including the mouse event handling logic and the address update logic during navigation. In the end, we discovered thirteen GUI spoofing bugs in IE 6, eleven of which were fixed when IE 7 was released.

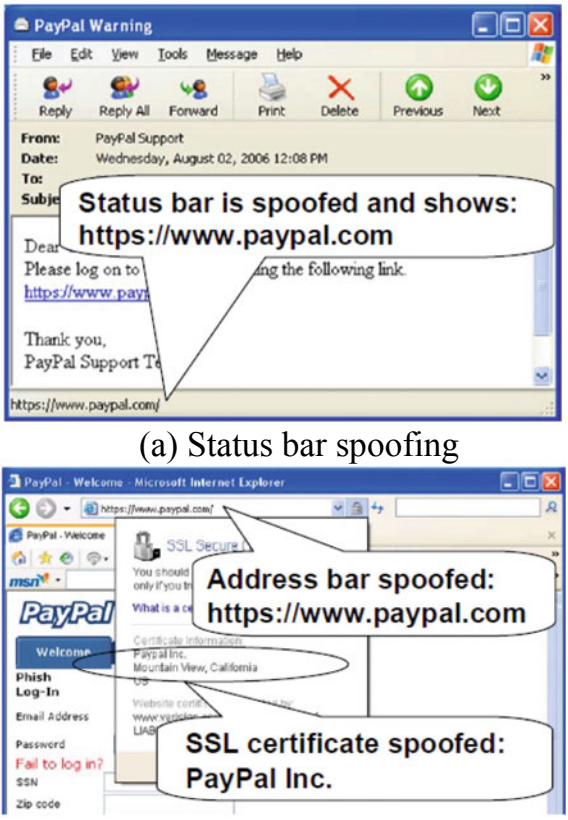

Fig. 3 Browser's GUI security bugs

3.2 Formal Methods for Authentication Protocols

Formal verification was the core technology in my research project that explicated the security assumptions of authentication SDKs. Major cloud providers, such as Facebook, Google, and Microsoft, provide single-sign-on authentication services (SSO) for website developers to integrate. With SSO, a website does not need to implement its own authentication infrastructure, but only needs to call an SSO service that it trusts. The SSO service is called the identity provider, or IdP. The website is called the relying party, or RP. The security goal is for the RP to authenticate the client as "Alice", if the client is able to authenticate to the IdP as "Alice".

Identity provider companies release SSO SDKs, and publish developer's guides to show the sequence of steps to integrate them into website code. However, an important question remains: if developers follow the guides in reasonable ways, will the resulting applications be secure? Our study shows that the answer today is "No". Many apps built using the SDKs we studied have serious security flaws. This is

not due to direct vulnerabilities in the SDK, but rather because achieving desired security properties by using an SDK depends on many implicit assumptions that are not readily apparent to app developers. These assumptions are not documented anywhere in the SDK or its developer documentation. In several cases, even the SDK providers are unaware of the assumptions.

The goal of our work [27] is to systematically identify the assumptions to use an SDK to produce secure applications. Our approach involves a combination of manual effort and automated formal verification. Any counterexample found by the verification tool indicates either (1) that our system models are not accurate, in which case we revisit the real systems to correct the model; or (2) that our models are correct, but additional assumptions need to be captured in the model and followed by application developers. The explication process is an iteration of the above steps so that we document, examine and refine our understanding of the underlying systems for an SDK. In the end, we get a set of formally captured assumptions and a semantic model that allow us to make meaningful assurances about the SDK: an application constructed using the SDK following the documented assumptions satisfies desired security properties.

The formal language we used in this study is Boogie [28]. It is an imperative language, so translating SDK code in a web language (e.g., PHP or C#) to Boogie is straightforward. Figure 4 shows an example PHP function translated into Boogie. The Boogie language allows the programmer to add assertions and invariants. The Boogie verifier will then statically verify whether the assertions and invariants hold in all circumstances.

```
protected function getUserFromAvailableData() {
  if ($signed_request) {
    ...
    $this->setPersistentData('user_id',
                             $signed_request['user_id']);
    return 0;
  }
  $user = $this->getPersistentData('user_id', $default = 0);
  $persist_token =
            $this->getPersistentData('access_token');
  $access_token = $this->getAccessToken();
  if ($access_token &&
        !($user && $persist_token == $access_token)) {
          $user = $this->getUserFromAccessToken();
          if ($user)
            $this->setPersistentData('user_id', $user);
          else $this->clearAllPersistentData();
    }
    return $user;
}
public function getLogoutUrl() {
  return $this->getUrl(
      'www',   'logout.php',
       array_merge(array(
         'next' => $this->getCurrentUrl(),
         'access_token' => $this->getAccessToken(), ), ...));
  }
```

```
procedure   {:inline 1} getUserFromAvailableData() returns (user:User) {
 if (IdP_Signed_Request_Records__user_ID[signed_request]!= _nobody)
   {   ...
     user := IdP_Signed_Request_Records__user_ID[signed_request];
     call setPersistentData__user_id(user);
     return;
   }
   call user := getPersistentData__user_id();
   call persisted_access_token := getPersistentData__access_token();
   call access_token := getAccessToken();
   if (access_token >= 0 &&
     !(user != _nobody && persisted_access_token == access_token)) {
         call user := getUserFromAccessToken(access_token);
         if (user != _nobody) {
             call setPersistentData__user_id(user);
         } else {
             call clearAllPersistentData();
         }
   }
   return;
}
procedure {:inline 1} getLogoutUrl()
  returns (API_id: API_ID, next__domain: Web_Domain, next__API:
         API_ID,   access_token: int)   {
             API_id := API_id_FBConnectServer_login_php;
             call access_token := getAccessToken();
             call   next__domain,   next__API := getCurrentUrl();
}
```

Fig. 4 Example of a PHP function and its Boogie model

4 Distributed Consensus

Another important topic that are originated in the dependability community and later plays an important role in security is distributed consensus. A legendary paper about this topic is the paper titled "The Byzantine General Problem" [29] by Lamport, Shostak and Pease. Despite the interesting title, the core problem studied in this paper is first presented in an earlier paper by the same set of authors. The earlier paper is titled "Reaching Agreement in the Presence of Faults" [30], which is clearly set in the fault tolerance context. The protocols proposed in reference [29] and later papers establish a well-known area in dependability and distributed systems, namely BFT (or Byzantine Fault Tolerance).

The fault tolerance capability of BFT comes from the redundancy of the nodes. However, it is different from other redundancy-based fault tolerance mechanisms. For example, triple modular redundancy (TMR) runs three replicas for a computation, and uses the majority voting to decide the output. TMR tolerates one faulty replica. However, TMR needs to have a component to do the voting. Although it can be substantially simpler than every replica, it can be faulty itself. Unfortunately, TMR cannot tolerate the faults of the voting component. BFT, on the other hand, does not assume any component to be reliable. Instead, the assumption of BFT is about the total number of the faulty components.

From fault tolerance to security. BFT and other distributed consensus protocols play an important role to ensure reliability of cloud platforms. However, it was somewhat unexpected that decentralized computing became the big wave of technological innovations. In this wave, consensus protocols are no longer a reliability mechanism, but form the foundation of decentralized trust.

Computing with decentralized trust enables scenarios that were hard to imagine before. For example, it was surprising that this new computing paradigm could be deployed in the global scale, enabling the worldwide community to issue a new currency (e.g., Bitcoin) without trusting any central bank. It was even more surprising that the new paradigm supported general purpose computing (e.g., by smart contracts on Ethereum). Decentralized trust enables many exciting possibilities, but its core mechanism is distributed consensus.

Consensus protocols can be categorized into two categories. The first is suitable for communities with open membership, which allow everyone to join. The Bitcoin network and the public Ethereum network are such communities. The second category is suitable for consortiums, which are formed by entities with clear identities.

Proof-of-Work (PoW) [31] is a representative consensus protocol in the first category. The goal of the protocol is to ensure that no member can dominate community. However, the open membership makes it impossible to base the protocol on identities, because a member can create an arbitrary number of identities. The core idea of PoW is to base the consensus on every member's actual computational power, which cannot be arbitrarily created. The PoW mechanism in Bitcoin is illustrated in Fig. 5. Each block contains the SHA256 hash value of its previous block, as well as the transactions contained in the current block. In addition, there is a nonce value, which

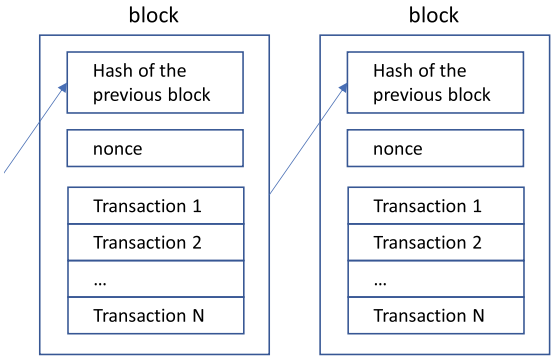

Fig. 5 Proof of work in Bitcoin

is crucial to the PoW mechanism. Any member who wants to construct a valid block to be accepted by the community (i.e., to represent the community consensus) needs to find such a nonce value that makes the hash of the current block begin with a predetermined number of zero bits. Because there is no known algorithm to calculate such a value efficiently, the only way to obtain it is to repeatedly try different value and calculate SHA256. This means that the probability for a member to represent the consensus is proportional to its actual computational power.

Besides the community with open membership, the other type of community is consortium, which consists of members with well-known identities. For example, government agencies, companies, and international organizations can form consortiums. In the consortium setting, traditional consensus protocols are valid. Xiao et al. provide a survey about blockchain consensus protocols, including those used in the consortium settings [32]. The survey covers Byzantine fault tolerant (BFT) protocols and crash fault tolerant (CFT) protocols. It categorizes protocols by different synchrony assumptions. Popular protocols include Raft [33], PBFT [34] and a few others. In 2019, Facebook announced the project to build a consortium blockchain called Libra. The consensus protocol, namely HotStuff [35], is derived from BFT.

5 Summary

Dependability (fault tolerance) is an important aspect of trustworthy computing. It is about investigating adversarial circumstances of a system and designing a mechanism for a system to be robust despite these circumstances. It is often distinguishable from security, for which the adversarial circumstances are intentionally created or controllable by a human attacker. For this reason, the adversary assumptions for security research often appear to be more direct and imminent. They are often deterministic, while the assumptions for dependability research are often probabilistic.

However, both research areas are fundamentally about quality of programming, thoroughness of testing, and good redundancies in design. My research career began in the dependability community. Then I worked on projects about security consequences of faults, later focused on security research. I realize that it is important for researchers to appreciate the commonality of the two disciplines. Among the three areas described in this chapter, the strong relevance of the bit-flip adversary and the distributed consensus was not foreseeable when I was initially exposed to the concepts. There was a decades-long research history of each topic in the dependability community, but the research value was revived stronger than before in the security community in the last 5 years. Regarding formal methods, I was not surprised by their values in security. It was reasonable to anticipate that logic-proof-based approaches would be needed to complement traditional software testing approaches. However, it is still very impressive to see a great amount of new formal techniques indeed make concrete contributions in many security domains.

References

1. Boneh D, DeMillo RA, Lipton RJ (1997) On the importance of eliminating errors in cryptographic computations. In: Proceedings of advances in cryptology: Eurocrypt'97, pp 37–51
2. Xu J, Chen S, Kalbarczyk Z, Iyer RK (2001) An experimental study of security vulnerabilities caused by errors. In: IEEE international conference on dependable systems and networks (DSN), Göteborg, Sweden
3. Chen S, Xu J, Iyer RK, Whisnant K (2002) Modeling and analyzing the security threat of firewall data corruption caused by instruction transient errors. In: IEEE international conference on dependable systems and networks (DSN), Washington DC
4. Govindavajhala S, Appel AW (2003) Using memory errors to attack a virtual machine. In: Proceedings of the IEEE symposium on security and privacy
5. Cowan C, Pu C, Maier D, Hinton H, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q (1998) Automatic detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX security symposium, San Antonio, TX
6. Baratloo A, Tsai T, Singh N (2000) Transparent runtime defense against stack smashing attacks. In: Proceedings of USENIX annual technical conference
7. Feng H, Giffin J, Huang Y, Jha S, Lee W, Miller B (2004) Formalizing sensitivity in static analysis for intrusion detection. In: Proceedings of the 2004 IEEE symposium on security and privacy
8. Forrest S, Hofmeyr S, Somayaji A, Longsta T (1996) A sense of self for Unix processes. In: Proceedings of the IEEE symposium on security and privacy
9. Feng H, Kolesnikov O, Fogla P, Lee W, Gong W (2003) Anomaly detection using call stack information. In: Proceedings of the IEEE symposium on security and privacy
10. Gao D, Reiter M, Song D (2004) Gray-box extraction of execution graphs for anomaly detection. In: Proceedings of the 11th ACM conference on computer and communication security
11. Giffin J, Jha S, Miller B (2004) Efficient context sensitive intrusion detection. In: Proceedings of the symposium on network and distributed system security
12. Hofmeyr SA, Forrest S, Somayaji A (1998) Intrusion detection using sequences of system calls. J Comput Secur 6(3)
13. Sekar R, Bendre M, Dhurjati D, Bollineni P (2001) A fast automaton-based method for detecting anomalous program behaviors. In: Proceedings of the IEEE symposium on security and privacy

14. Crandall JR, Chong FT (2004) Minos: control data attack prevention orthogonal to memory model. In: Proceedings of the 37th international symposium on microarchitecture
15. Smirnov A, Chiueh T (2005) DIRA: automatic detection, identification and repair of control-data attacks. In: Proceedings of the 12th network and distributed system security symposium (NDSS), San Diego, CA
16. Suh G, Lee J, Devadas S (2004) Secure program execution via dynamic information flow tracking. In: Proceedings of the 11th international conference on architectural support for programming languages and operating systems. Boston, MA
17. Andersen S, Abella V, Data execution prevention. Changes to functionality in Microsoft Windows XP service pack 2, part 3: memory protection technologies. http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx
18. Otachi E. What is data execution prevention in Windows 10. https://helpdeskgeek.com/windows-10/what-is-data-execution-prevention-in-windows-10/
19. Chen S, Xu J, Sezer EC, Gauriar P, Iyer RK (2005) Non-control-data attacks are realistic threats. In: Proceedings of USENIX security symposium
20. Kim Y, Daly R, Kim J, Fallin C, Lee JH, Lee D, Wilkerson C, Lai K, Mutlu O (2014) Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. In: Proceedings of the international symposium on computer architecture (ISCA)
21. Cojocar L, Razavi K, Giuffrida C, Bos H (2019) Exploiting correcting codes: on the effectiveness of ECC memory against Rowhammer attacks. In: Proceedings of the IEEE symposium on security and privacy
22. Cojocar L, Kim J, Patel M, Tsai L, Saroiu S, Wolman A, Mutlu O (2020) Are we susceptible to Rowhammer? An end-to-end methodology for cloud providers. In: Proceedings of the IEEE symposium on security and privacy
23. Rosu G, Chen F (2003) Certifying measurement unit safety policy. In: Proceedings of the IEEE international conference on automated software engineering (ASE)
24. Ball T, Cook B, Levin V, Rajamani SK, SLAM and static driver verifier: technology transfer of formal methods inside Microsoft. Microsoft Research Technical Report MSR-TR-2004-08
25. Chen S, Meseguer J, Sasse R, Wang HJ, Wang Y-M (2007) A systematic approach to uncover security flaws in GUI Logic. In: Proceedings of the IEEE symposium on security and privacy
26. Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N et al (2002) Maude: specification and programming in rewriting logic. Theor Comput Sci 285(2):2002
27. Wang R, Zhou Y, Chen S, Qadeer S, Evans D, Gurevich Y (2013) Explicating SDKs: uncovering assumptions underlying secure authentication and authorization. In: Proceedings of the USENIX security symposium
28. Boogie: an intermediate verification language. http://research.microsoft.com/en-us/projects/boogie/
29. Lamport L, Shostak R, Pease M (1982) The Byzantine generals problem. ACM transactions on programming languages and systems
30. Pease M, Shostak R, Lamport L (1980) Reaching agreement in the presence of faults. J ACM
31. Nakamoto S, Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf
32. Xiao Y, Zhang N, Lou W, Thomas Hou Y (2020) A survey of distributed consensus protocols for Blockchain networks. In: IEEE communications surveys & tutorials, vol 22
33. Ongaro D, Ousterhout J (2014) In search of an understandable consensus algorithm. In: 2014 USENIX annual technical conference (USENIX ATC 14), pp 305–319
34. Castro M, Liskov B (1999) Practical byzantine fault tolerance. In: Proceedings of symposium on operating systems design and implementation (OSDI)
35. Yin M, Malkhi D, Reiter MK, Gueta GG, Abraham I, HotStuff: BFT consensus in the lens of Blockchain. [arXiv:1803.05069] https://arxiv.org/pdf/1803.05069.pdf