Catastrophic Cyber-Physical Malware*

Suresh Kothari[1, 2], Ganesh Ram Santhanam[1], Payas Awadhutkar[1], Benjamin Holland[1], Jon Mathews[2], and Ahmed Tamrawi[2]

[1]Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011
[2]EnSoft Corp., 2625 N Loop Dr Suite 2580, Ames, IA 50010

Abstract

With the advent of highly sophisticated cyber-physical malware (CPM) such as Industroyer, a cyber-attack could be as destructive as the terrorist attack on 9/11, and it would virtually paralyze the nation. We discuss as the major risks the vulnerability of: telecommunication infrastructure, industrial control systems (ICS), and mission-critical software.

In differentiating CPM from traditional malware, the difference really comes from the open-ended possibilities for malware triggers resulting from the wide spectrum of sensor inputs, and the almost limitless application-specific possibilities for designing malicious payloads.

Fundamentally, the challenges of detecting sophisticated CPM stem from the complexities inherent in the software at the heart of cyber-physical systems. We discuss three fundamental challenges: explosion of execution behaviors, computational intractability of checking feasible behaviors, and difficult-to-analyze programming constructs.

In detecting novel CPM, the tasks are: developing plausible hypotheses for malware trigger and malicious payload, analyzing software to gather evidence based on CPM hypotheses, and verifying software to prove or refute a hypothesis based on the gathered evidence. We discuss research directions for effective automation to support these tasks.

1 Introduction

The imminent danger of cyber-physical malware (CPM) is evident from attacks such as the power outage in Ukraine [122] or the hijacking of a Jeep Cherokee [83]. The net-centricity of modern systems offers an adversary affordable attack vectors through cyberspace against critical missions. We are arguably at risk to an asymmetric attack vector launched by a terrorist organization or rogue nation that cannot, or chooses not to confront in a conventional conflict. The Internet of Things (IoT) implies more software-driven devices and thus increased CPM risk.

The traditional notion of malware is too narrow, and the prevalent characterizations (virus, worm, Trojan horse, spyware etc.) are neither precise nor comprehensive enough to characterize CPM. Detecting sophisticated CPM is like searching for a needle in the haystack without knowing what the needle looks like. Employing real-world examples, this survey chapter discusses: the fundamentals of CPM, the need for threat modeling, analysis and verification of CPM, and the challenges and directions for future research.

CPS security problems are often rooted in the complex CPS software. Securing CPS software requires knowledge of both software analysis and verification as well as CPS architecture and attack surface. It is hard for the CPS community to understand the intricacies of software analysis and verification. Whereas for the software engineering community, the lack of adequate CPS knowledge is a major roadblock. Technological advances in computing, communications, and control have set the stage for a next generation of CPS for energy, environment, transportation, and health care. The context for modeling CPM [101, 104, 58] needs to

*This material is based on research sponsored by DARPA under agreement numbers FA8750-15-2-0080 and FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

be exposed so that the software engineering community can engage in collaborative interdisciplinary research for evolving CPM characterizations that can cover the vast expanse from mobile phones apps to Supervisory Control and Data Acquisition (SCADA) systems.

Let us overview testing and verification as techniques for software assurance. Testing cannot verify that every potential vulnerability instance is safe or not. Avionics companies try to compensate for limitations of testing by requiring high test coverage using the modified condition/decision coverage (MC/DC) metric [45] and instituting stringent software development and auditing practices as required by DO-178B [45]. When it comes to cybersecurity, the limitations of testing become more pronounced. An attacker can craft a clever trigger for the malicious software to defy getting caught with high test coverage.

Next, let us consider formal verification [116, 59] as an alternative to testing. The point often argued in its favor is that it can verify whether every instance is safe or not. The end result of formal verification can be: (a) it proves that an instance is safe, (b) it proves that an instance is unsafe, or (c) it is inconclusive when it crashes or times out. Formal verification provides a counter example as evidence for (b). However, it does does not provide evidence for (a) and (c). The core dump it may provide for (c) is not human-comprehensible. While scalability of formal verification has been the subject of intense research with the use of *binary decision diagrams* and host of other techniques [80], the topic of automated verification with human-comprehensible evidence has not received much attention. Avionics, automotive and other industry practitioners of safety-critical software consider such lack of evidence a serious short-coming of formal methods [64]. De Millo, Lipton, and Perlis (the first Turing Award recipient) [70] have argued that software verification, like "proofs" in mathematics, should provide *evidence* that humans can follow and thus be able to build trust into the correctness of the software verification. This is especially crucial given the potential for tremendous harm from CPM.

With novel CPM, the first challenge is to hypothesize it. The sensor inputs create open-ended possibilities for attackers to craft CPM. The analyst must narrow down a nebulous specification of a vulnerability to something specific that can be verified. Unbeknownst to the user, a global positioning system (GPS) may contain malicious code that compromises integrity of the system. The analyst must specifically hypothesize how the integrity breach could occur. Currently, threat modeling for trigger-based CPM is often a tedious manual endeavor with hardly any automated tools support [56].

A completely automated solution for detecting catastrophic malware in mission-critical software is unlikely, as it is an extremely complex problem. Fred Brooks (1996 Turing Award recipient) points out [55]: "If indeed our objective is to build computer systems that solve very challenging problems, my thesis is that IA > AI, that is, that intelligence amplifying systems can, at any given level of available systems technology, beat AI systems. That is, a machine and a mind can beat a mind-imitating machine working by itself." As elaborated in our paper [93], there is dire need for analysis and verification tools to facilitate a human-in-the-loop approach for addressing the CPM problem.

This survey chapter leverages our team's experience with: (a) analyzing complex CPM on the DARPA Automated Program Analysis for Cybersecurity (APAC) [5] and Space/Time Analysis for Cybersecurity (STAC) [29] programs, and (b) designing and developing commercial products to model and analyze control systems software for automobile, avionics, and other industries for whom safety and security is a major concern.

2 CIA Triad and CPM Metrics

Government agencies such as National Institute of Standards and Technology (NIST) and National Security Agency (NSA) have channelized their efforts towards developing metrics for measuring the ease of exploitability, and the impact of CPM (e.g., national vulnerability database (NVD) [24] and common vulnerability scoring system (CVSS) [12]). These efforts to calibrate CPM are aimed at enabling industry and government to better assess and manage risks.

2.1 CIA Triad

The CIA triad (confidentiality, integrity, availability) [22] has evolved as a general and robust model to systematically explore hypotheses related to malicious behaviors. CIA triad covers a vast expanse of CPM

that are targeted towards devices ranging from smart phone apps to power grids. Thus, the CIA triad may be used as a framework to hypothesize potential vulnerabilities.

For example, confidentiality may be breached in an Android app when sensitive data (e.g., image data from the camera, or GPS location) is leaked to an unauthorized sink (e.g., the internet, or to an adversary). Similarly, integrity may be breached in a GPS Android app when the coordinates are incorrectly shown in some geographic locations. Availability may be breached in a text processing application if the application runs vulnerable sorting algorithms that have asymmetrically large runtime on certain relatively small inputs (e.g., quick sort whose pivot degenerates to the last element in the unsorted input list for certain cases, or the app runs unnecessary loops to drain the device battery). In some cases, sophisticated CPM may breach two or all three CIA security attributes. For example, Stuxnet [31] was able to access and modify sensitive information (breach confidentiality and integrity) about Siemens PLC controllers installed on the centrifuges while remaining unobservable for years. This eventually led to damage of the centrifuge controllers beyond recovery (breach of availability).

2.2 Metrics for Measuring CPM Impact

We mention a few of the prevalent metrics that measure the impact of attacks caused by CPM on ICS. For a detailed discussion of CPM impact metrics, see [65].

- The *CVSS* measures CPM impact in terms of three kinds of metrics (See [100]): (a) *Base* metrics measure how easy it is for an attacker to exploit the vulnerability, and the confidentiality, integrity and availability impact of the vulnerability on the compromised system. (b) *Temporal* metrics represent the evolving exploitability of a vulnerability, such as the availability of exploit code and availability of patches to fix the vulnerability. (c) *Environmental* metrics represent the characteristics of a vulnerability that are specific to a particular user's environment, such as the potential damage incurred by the organization due to CIA breach.

- *Potential impact metrics* is part of a framework drafted by the National Security Agency (NSA) for ICS networks, outlining potential impact and loss due to a CPM attack on a system. This framework also characterized CPM attack impact in terms of the CIA triad.

- *Ideal-based metrics* defined seven security dimensions and the ideal or best possible values for each of them [54]. The seven dimensions include: Security Group Knowledge, Attack Group Knowledge, Access, Vulnerabilities, Damage Potential, Detection, and Recovery. For example, the ideal for the Vulnerabilities dimension is that there are no vulnerabilities in the system, and the ideal for the Damage Potential dimension is that there is no confidentiality, integrity or availability impact to the system in the face of a CPM attack. The ideals are meant as a reference point to assess a system's vulnerability, and are not necessarily realizable in practice.

A common theme that runs across all prevalent CPM attack impact metrics is that they characterize impact in terms of the CIA-triad: (a) confidentiality breach (leak of sensitive information to an adversary), (b) integrity breach (corruption of sensitive information by an adversary), and (c) availability breach (denial of service to legitimate users due to excessive consumption triggered by CPM).

3 CPM Attack Phases

A cyber attack is not all that different from a military attack. A cyber attacker will dedicate a significant amount of time observing and probing the target organization to find weaknesses in its defense. Any weakness found may lead to infiltration and eventually an assault. We have consolidated the discussion of cyber attack phases into three main phases. We have used the United States Navy Academy [32] as the primary source of information for this discussion.

3.1 Reconnaissance

An attacker's first goal is to identify potential targets for their mission. Attackers are often motivated by financial gain, access to sensitive information or damage to an entity (could be a company, organization, nation etc.).

The attacker may collect information on targeted organization's security systems and available entry points. The attacker might set up a fake company, register domains and create fake profiles for social engineering purposes.

Once the attacker determines what defenses are in place, the next step is to identify a weak point that allows the attackers to gain access. This is usually accomplished by scanning an organization's network with tools easily found on the Internet. This step of the process usually proceeds slowly, sometimes lasting months, as the attackers search for vulnerabilities. We list some critical information that are typically obtained during the reconnaissance phase: (a) Network Information: IP Addresses, subnet mask, network topology, domain names; (b) Host Information, user names, group names, operating system family and version, TCP and UDP services; (c) Security Policies: password complexity requirements, password change frequency, expired/disabled account retention, physical security, firewalls, intrusion detection systems; and (d) Human Information: home address, telephone number, frequent hangouts (physical and online), computer knowledge, hobbies and interests. Based on how interaction is done with the target subject, the reconnaissance can be passive and active.

Passive Reconnaissance is gathering information in a manner unlikely to alert the subject of the surveillance. This is the natural start of any reconnaissance because, once alerted, a target will likely react by drastically increasing security in anticipation of an attack. The attacker minimizes any interaction with the target network which may raise flags in the computer logs. For example, visiting the target's website may leave behind a trace that your IP Address established a TCP connection to the target's web server, but it will be one of millions of connections that day - probably not going to stand out to the administrator in the periodic review of server logs. On the other hand, visiting the target's website so frequently that the server becomes overloaded is certain to alert an administrator.

Active Reconnaissance is gathering information while interacting with the subject directly, in a way that usually can be discovered. A number of tools that can be used for active network recon: ping, traceroute, and netcat (nc). If the attackers know the range of potential IP addresses for their target network, they can use ping to determine which IPs are actually in use by hosts on the network. They can use traceroute to figure out the topology of the network: i.e. where the routers are with respect to the hosts. Finally, they can use netcat (nc) to determine which ports are open with servers listening on them. Nmap is a powerful network scanner that attackers use to discover hosts on a target network.

As a defensive measure, an organization should centrally collect the log messages related to established or rejected connections from their network devices and use a tool to visualize their network communications and connection paths.

3.2 Intrusion and Escalation

At the second phase of a cyber attack, the attacker seeks to breach the organization perimeter and gain a persistent foothold in the environment.

Now that weaknesses in the target network are identified, the next step in the cyber attack is to gain access and then escalate. In almost all such cases, privileged access is needed because it allows the attackers to move freely within the environment. Techniques and tools such as Rainbow tables help intruders steal credentials, escalate privileges to admin. Once the attackers gain elevated privileges, the network is effectively taken over by the attackers. The attackers can spear-phish the company to gain credentials, use valid credentials to access the corporate infrastructure, and download more tools to access the environment. The initial intrusion is expanded to persistent, long-term, remote access to the environment. Once the attackers own the target organization's network, they establish a command and control channel from the outside into the victim's infrastructure.

As a defensive measure, it is important to have an enterprise-wide log management. It is important to constantly monitor network traffic and look for anomalies and signs of attacks, and to make intrusion harder, add two factor authentication to the services. The goal is to detect and disarm the control channel before

the attacker can start to move laterally inside the network, causing more harm. One can use network and operating system logs to find connections from the outside that should not be there.

3.3 Assault

The final stage is where cost to businesses rise exponentially if the attack is not defeated. This is when the attacker executes the final aspects of their mission, stealing intellectual property or other sensitive data, corrupting mission-critical systems, and generally disrupting the operations of the victim's business.

This is when the hackers might alter the functionality of the victim's application, or disable the application entirely. Typically, attackers use exploit kits [95] which use drive-by downloads to download and run the appropriate exploit for the target system. The Stuxnet attack [31] on Iran's critical infrastructure is a classic example. During the assault phase, the attack ceases to be stealthy. However, the attackers have already effectively taken control of the environment, so it is too late for the breached organization to defend itself. Finally, the attacker may either terminate the connection if no further access is required, or create a backdoor for future access to the target.

Usually the attackers want to hide their tracks, but this is not universally the case, especially if the hackers want to leave a calling card behind to boast about their exploits. The purpose of trail obfuscation is to confuse, disorientate and divert the forensic examination process. Trail obfuscation covers a variety of techniques and tools including log cleaners, spoofing, misinformation, backbone hopping, zombied accounts, trojan commands, and more. The "Flame" malware [67] came to light in the summer of 2012. It's a very sophisticated piece of malware, probably produced by some nation-state, not by random hackers, terrorists or criminals. One of the many interesting aspects of the Flame malware was that it was designed to "cover its tracks", i.e. to erase traces of its existence on computers that it had infected, but was finished with. The creators of the Flame cyber-espionage threat ordered infected computers still under their control to download and execute a component designed to remove all traces of the malware and prevent forensic analysis [67]

Note that not all CPM attacks necessarily go through all the above phases. For example, side channel attacks do not require any installation and command or control; passive observations suffice for that.

4 National Cybersecurity: Critical Concern and Need

When people think of cybersecurity today, they worry about hackers and criminals who prowl the Internet, stealing people's identities, sensitive business information, or even national security secrets. Those threats are real and they exist today. But the even greater danger facing us in cyberspace goes beyond crime and harassment. A cyber attack perpetrated by nation states or violent extremists groups could be as destructive as the terrorist attack on 9/11. Such a destructive cyber-terrorist attack could virtually paralyze the nation. The most destructive scenarios involve cyber-terrorists launching several attacks on our critical infrastructure at one time, in combination with a physical attack. Attackers could also seek to disable or degrade critical military systems and communication networks. The collective result of these kinds of attacks could be a cyber Pearl Harbor: an attack that would cause physical destruction and loss of life. In fact, it would paralyze and shock the nation and create a new and profound sense of vulnerability. These are observations of the Secretary of Defense Leon Panetta [103].

Internet of Things (IoT) implies increasing dependence on software, making software assurance an important requirement for everyday life. White House reports [37] point to the urgent national need to shift the cybersecurity posture from defending computer networks to assuring critical missions. Telecommunication infrastructure, industrial control systems and mission-critical software are critical concerns for the security of cyberphysical systems.

4.1 Risk: Telecommunication Infrastructure

Telecommunications hardware includes a vast range of products that enable communication across the entire planet, from video broadcasting satellites to telephone handsets to fiber-optic transmission cables. Services include running the switches that control the phone system, providing Internet access, and configuring private networks by which international corporations conduct business. Software makes it all work, from sending

and receiving e-mail to relaying satellite data to controlling telephone switching equipment to reducing background noise on your cell phone call.

Huawei, a Chinese multinational, is the largest telecommunications equipment manufacturer in the world in 2017, having overtaken Ericsson in 2012. In June 2016, Huawei was reportedly working on and designing its own mobile OS for future usage. From July to September 2017, Huawei surpassed Apple and became the second largest smartphone manufacturer in the world after Samsung. It also aims to be one of the world's five largest cloud players in the near future. Currently, the world's five largest vendors of telecommunication equipment (excluding mobile phone handsets), ranked by revenues are: Huawei, Ericsson, Cisco, Nokia (including Alcatel-Lucent), and ZTE corporation [33].

Warning about a potential threat to national security, US lawmakers decided to scrutinize a bid by Huawei to supply telecommunications equipment to Sprint Nextel in the United States. Sprint Nextel reportedly decided in 2011 to block Chinese companies Huawei and ZTE from its multi-billion-dollar network modernization project because of mounting national security concerns. The US House Intelligence Committee conducted an investigation of Huawei, recommending that the US government should block acquisitions, takeovers, or mergers involving Huawei, given the threat to US national security interests [40]. Committee Chairman Mike Rogers noted claims by US companies that Huawei equipment exhibited unexpected behavior, including routers allegedly sending large data packets to China late at night.

The telecommunications sector plays a critical role in the safety and security of a nation, and is thus a target of foreign intelligence services. The country's reliance on telecommunications infrastructure includes more than consumers' use of computer systems. Multiple critical infrastructure systems depend on information transmission through telecommunications systems. These modern critical infrastructures include electric power grids; banking and finance systems; natural gas, oil, and water systems; and rail and shipping channels. Inter-dependencies among these critical infrastructures greatly increase the risk that failure in one system will cause failures or disruptions in multiple critical infrastructure systems. Therefore, a disruption in telecommunication networks can have devastating effects on all aspects of modern living, causing shortages and stoppages that ripple throughout society.

A company providing telecommunication equipment is likely to have access to or detailed knowledge of the telecommunication infrastructures' architectural blueprints. The threat posed to national security interests by vulnerabilities in the telecommunications supply chain is an increasing priority given the country's reliance on interdependent critical infrastructure systems; the range of threats these systems face, the rise in cyber espionage, and the growing dependence all consumers have on a small group of equipment providers.

4.1.1 Complexity of Network Configuration

Effective and efficient configuration of networks has been widely recognized as a grand challenge [97]. It is a challenge to analyze how settings in the configuration space impact functionality and security. Misconfigurations are prevalent and can have a dramatic impact. For example, one misconfigured router in AS 9121 (Autonomous System in Turkey) resulted in misdirected/lost traffic for tens of thousands of networks [105]. In real-world systems, the size of configurations is large and can easily reach thousands of lines of commands in each router, while there are hundreds to thousands of routers in a large network. Finally, network configurations are decomposed in multiple routers as in distributed programs, and these distributed pieces are dependent upon one another. Configuring routers is a tedious, error-prone and complex task. The paper [125] presents a quantitative study of configuration errors in 37 firewalls.

In addition to the complexity of managing a network of firewalls, the process of ensuring their correctness is even more complicated due to subtle interactions between firewall configurations and the dynamics of routing. Configuration problems occur between firewalls of different devices placed along a network path, and such a distributed problem might surface only in a particular routing state. To detect such inconsistencies, there is a need to consider routing as well as firewall configurations. Particularly in a large, growing network with a complex topology, keeping track of all of the possible sets of routes can be extremely time-consuming and inaccurate.

The complexity of the configuration process is analogous to distributed assembly programs [97, 77, 47]. For example, the behavior of inter-domain routing policy configurations could be modeled as a program flow graph similar to the way a compiler models a program. This graph represents the way routes are advertised according to the routing policy configurations in a network. The program flow graph can be used to do

what-if scenario testing for any pending changes to a configuration.

4.2 Risk: Industrial Control Systems

Those living in developed industrialized nations tend to take modern-day conveniences for granted. Flip a light switch, and the home illuminates. Turn on the tap, and clean water flows. It all happens routinely, without a hiccup. Now imagine a world where the delivery of sustained services is interrupted. Suddenly there is no clean water, or electric power disappears. The effects of such failures can be disastrous. How could such disasters happen? Industrial controls systems (ICS), the backbone of such services, can be attacked. Threats to ICS can come from numerous sources, including adversarial sources such as hostile governments, terrorist groups, industrial spies, disgruntled employees, malicious intruders, and natural sources such as from system complexities, human errors and accidents, equipment failures and natural disasters. The term "ICS," as used throughout this section, includes Supervisory Control and Data Acquisition (SCADA) systems.

On August 17, 2009, at the Sayano–Shushenskaya hydroelectric power station in Russia, excessive vibration caused a 920 ton turbine to break apart, flooding the facility, killing 75 people, and causing a power grid failure. It happened because the control software designed to shut down the turbine in the event of excessive vibration did not work. While commenting on the accident, General Keith B. Alexander, commander of U.S. Cyber Command noted that we are living in a time where such a deadly incident could also happen as a result of a cyber attack [48].

The December 2016 attack on the Ukrainian power grid points to the use of Win32/Industroyer, an advanced piece of devastating malware targeted at an ICS. Industroyer is considered the biggest threat to ICS since Stuxnet [42, 60]. Industroyer is highly customizable malware. It is universal, in that it can be used to attack any industrial control system using some of the targeted communication protocols. For example, its wiper component and one of the payload components are tailored for use against systems incorporating certain industrial power control products by ABB, and the DoS component works specifically against Siemens SIPROTECT devices used in electrical substations and other related fields of application [60].

Industroyer is modular malware (Figure 1). Its core component is a backdoor used by attackers to manage the attack: it installs and controls the other components and connects to a remote server to receive commands and to report to the attackers.

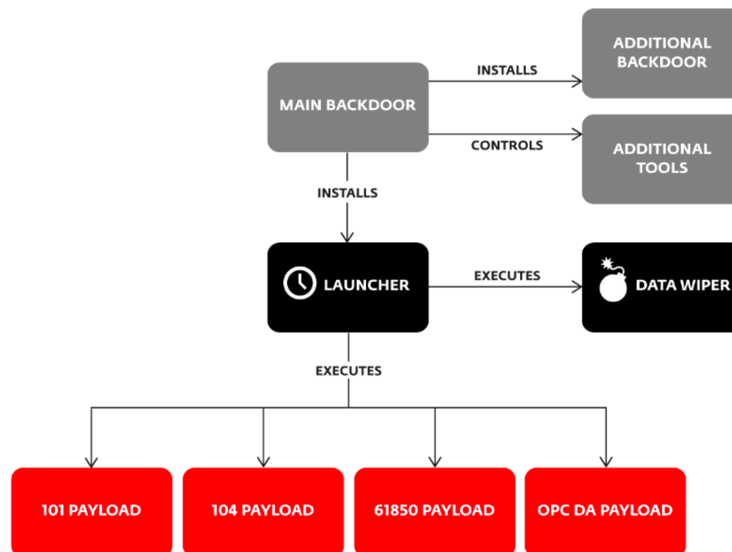


Figure 1: Schematic of Industroyer ICS malware

What sets Industroyer apart from other malware targeting infrastructure is its use of four payload components, which are designed to gain direct control of switches and circuit breakers at an electricity distribution substation. Each of these components targets particular communication protocols specified in the following

standards: IEC 60870-5-101, IEC 60870-5-104, IEC 61850, and OLE for Process Control Data Access (OPC DA).

Industroyer's dangerousness lies in the fact that it uses protocols in the way they were designed to be used. The problem is that these protocols were designed decades ago, and back then industrial systems were meant to be isolated from the outside world. Thus, their communication protocols were not designed with security in mind. That means that the attackers did not need to look for protocol vulnerabilities; all they needed was to teach the malware "to speak" those protocols.

The U.S. Department of Homeland Security (DHS) National Cyber Security Division's Control Systems Security Program (CSSP) performs cybersecurity assessments of ICS to reduce risk and improve the security of ICS and their components used in critical infrastructures throughout the United States [39].

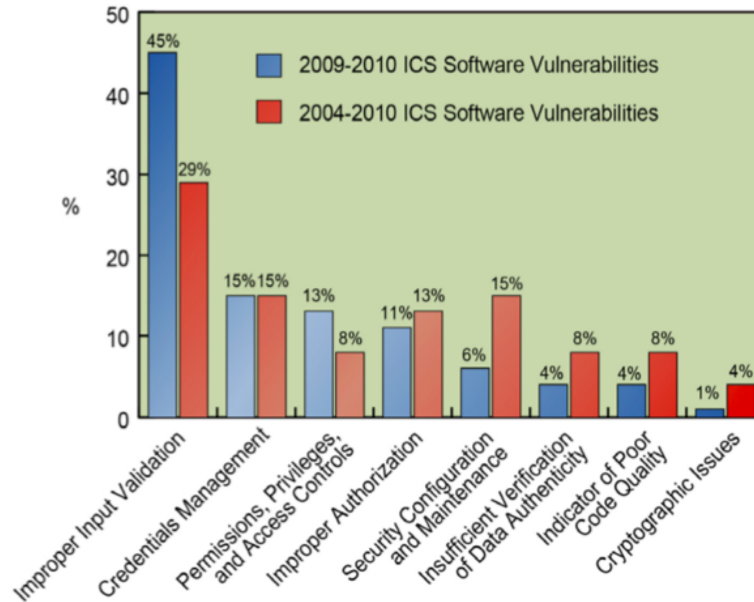


Figure 2: Changing landscape of ICS security vulnerabilities

Most ICS began as proprietary, stand-alone collections of hardware and software that were walled off from the rest of the world and isolated from most external threats. Today, widely available software applications, Internet-enabled devices and other non-proprietary IT offerings have been integrated into most such systems. This connectivity has delivered many benefits, but it also has increased the vulnerability of these systems. The changing landscape of ICS security vulnerabilities (Figure 2) based on data from DHS Cybersecurity Report [39]): the improper input validation vulnerabilities (e.g., buffer overflow) have gone down from 45% to 29%; whereas, the security configuration and maintenance vulnerabilities have shown a major up rise from 6% to 15%.

Cybersecurity Evaluation Tool (CSET) from DHS is a desktop software tool that guides users through a step-by-step question and answer process to collect facility-specific control and enterprise network information. CSET is a self-assessment software standards application for performing cybersecurity reviews of industrial control and enterprise network systems. The tool may be used by any organization to assess the cybersecurity posture of ICS that manage a physical process or enterprise network. The tool also provides information that assists users in resolving identified weaknesses in their networks and improving their overall security posture.

The NIST Guide to Industrial Control Systems Security [79] discuss three broad categories of ICS incidents including intentional attacks, unintentional consequences or collateral damage from worms, viruses or control system failures, and unintentional internal security consequences, such as inappropriate testing of operational systems or unauthorized system configuration changes. The NIST report discusses examples of intentional attacks such as Stuxnet[31] and Maroochy Shire Sewage Spill [115], and also examples of unintentional consequences such as Davis-Besse [106] and the Northeast Power Blackout [36].

Maroochy Shire Sewage Spill: In the spring of 2000, a former employee of an Australian organization that develops manufacturing software applied for a job with the local government, but was rejected. Over a two-month period, the disgruntled rejected employee reportedly used a radio transmitter on as many as 46 occasions to remotely break into the controls of a sewage treatment system. He altered electronic data for particular sewerage pumping stations and caused malfunctions in their operations, ultimately releasing about 264,000 gallons of raw sewage into nearby rivers and parks.

Stuxnet: Stuxnet is a Microsoft Windows computer worm discovered in July 2010 that specifically targets industrial software and equipment. The worm initially spreads indiscriminately, but includes a highly specialized malware payload that is designed to target only specific SCADA systems that are configured to control and monitor specific industrial processes.

Davis-Besse Nuclear Plant: In August 2003, the Nuclear Regulatory Commission confirmed that in January 2003, the Microsoft SQL Server worm known as Slammer infected a private computer network at the idled Davis-Besse nuclear power plant in Oak Harbor, Ohio, disabling a safety monitoring system for nearly five hours. In addition, the plant's process computer failed, and it took about six hours for it to become available again. Slammer reportedly also affected communications on the control networks of at least five other utilities by propagating so quickly that control system traffic was blocked.

Northeast Power Blackout: In August 2003, failure of the alarm processor in First Energy's SCADA system prevented control room operators from having adequate situational awareness of critical operational changes to the electrical grid. Additionally, effective reliability oversight was prevented when the state estimator at the Midwest Independent System Operator failed due to incomplete information on topology changes, preventing contingency analysis. Several key 345 kV transmission lines in Northern Ohio tripped due to contact with trees. This eventually initiated cascading overloads of additional 345 kV and 138 kV lines, leading to an uncontrolled cascading failure of the grid. A total of 61,800 MW load was lost as 508 generating units at 265 power plants tripped.

An IBM Report [88] published in 2015 looks at the history of ICS, the susceptibility of these systems to certain attacks, and how the systems can be defended. According to IBM Managed Security Services (MSS) data, attacks targeting ICS increased over 110 percent in 2016 [88]. Specifically, the spike in ICS traffic was related to SCADA brute-force attacks, which use automation to guess default or weak passwords. Once broken, attackers can remotely monitor or control connected SCADA devices. In January 2016, a penetration testing solution [22] containing a brute-force tool that can be used against Modbus [21], a serial communication protocol, was released. The public release and subsequent use of this tool by various unknown actors likely led to the rise in malicious activity against ICS in 2016 [88].

4.3 Risk: Vulnerable Mission-Critical Software

If a desktop operating system fails, the computer can be rebooted. If a flight control system fails, it can be a disaster with no chance to reboot the system. Malfunctioning of mission-critical software results in serious impact on business operations or upon an organization, and even can cause social turmoil and catastrophes. Mission-critical software drives online banking systems, railway/aircraft operating and control systems, electric power systems, and many other computer systems that adversely affect business and society when they fail.

There are four different types of critical systems: mission critical, business critical, safety critical and security critical. The key difference between a safety critical system and a mission critical system, is that a safety critical system is a system that, if it fails, may result in serious environmental damage, injury, or loss of life, while a mission critical system may result in failure in goal-directed activity. An example of a safety critical system is a chemical manufacturing plant control system. Mission critical system and business critical system are similar terms, but a business critical system fault can influence only a single company or an organization. A security critical system may lead to loss of sensitive data through theft or accidental loss.

4.3.1 Examples of Mission-Critical Software Flaws

Following are two concrete examples of mission-critical software to show how the nature and the code mechanics can vary significantly across flaws.

Ariane 5 Software Flaw: On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after lift-off. The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million. A board of inquiry investigated the causes of the explosion and in two weeks issued a report. It turned out that the cause of the failure was a software error in the inertial reference system. Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger than 32,767, the largest integer storeable in a 16 bit signed integer, and thus the conversion failed [98].

Apple SSL Flaw: In February 2014, Apple published iOS 7.0.6, a security update for its mobile devices. The update was a patch to protect iPhones, iPads and iPods against what Apple described as a *data security* problem: an attacker with a privileged network position may capture or modify data in sessions protected by SSL/TLS. SSL stands for Secure Sockets Layer and, it is the standard technology for keeping an internet connection secure and safeguarding any sensitive data that is being sent between two systems, preventing criminals from reading and modifying any information transferred, including potential personal details. In short, the software layer for secure connection itself was flawed and became leaky. The flawed Apple SSL code segment is shown below.

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  // Bug
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
. . .
```

The following quote from an analysis of the bug by the security firm Sophos [76] provides a concise summary of what went wrong.

> The programmer is supposed to calculate a cryptographic checksum of three data items via the three calls to SSLHashSHA1.update(), and then to call the all important function sslRawVerify(). If sslRawVerify() succeeds, then err ends up with the value zero, which means "no error", and the SSLVerifySignedServerKeyExchange function returns to say, "all good." But in the middle of this code fragment, you can see that the programmer has accidentally (no conspiracy theories, please!) repeated the line "goto fail;". The first goto fail happens if the if statement succeeds, i.e. if there has been a problem and therefore err is non-zero. This causes an immediate "bail with error," and the entire TLS connection fails. But because of the peccadilloes of C, the second goto fail, which should not be there, always happens if the first one does not, i.e. if err is zero

and there is actually no error to report. The result is that the code leaps over the vital call to sslRawVerify(), and exits the function. This causes an immediate "exit and report success," and the TLS connection succeeds even though the verification process has not actually taken place.

In short, the addition of second `goto fail;` statement following an `if` statement without curly braces indicates an unconditional control flow jump to the `fail` block making the code following the second `goto fail;` statement unreachable. The result was that critical signature checking code not executed, allowing invalid certificates to by quietly accepted as valid signatures. The bug went undetected for nearly a year, affecting both personal computers and mobile devices.

4.3.2 Dormancy of Mission-Critical Software Flaws

Flaws in mission-critical software can remain dormant and be silently exploited. The dormancy is especially troubling because the damage caused by the flaws can go undetected while the damage continues. In Apple SSL and Heartbleed examples the critical secure communication features themselves were flawed and caused serious confidentiality breaches without being noticed. In the Stuxnet example, the malware has actually two elements, the first one caused a confidentiality breach. The second element caused an integrity breach in the control system. The second malware element silently destroyed almost a fifth of Iran's reactors. The first malware element enabled espionage to learn the electrical blueprint of the reactors. And the first element of Stuxnet was only detected with the knowledge of the second.

The Apple SSL flaw appears to have been introduced in a code change made ahead of the launch of iOS 6.0. It became public and Apple released a fix 15 months later in iOS 7.0.6. The flaw also existed in Mac OS X and for which the fix came even later. Depending on who knew about it, it allowed connections to secure sites to be spied on and/or login details captured. In other words, all iOS and Mac OSX users were subject to a serious confidentiality breach for several months.

In his blog [114], the noted cybersecurity expert Bruce Schneier commented on the Heartbleed flaw "Catastrophic is the right word. On the scale of 1 to 10, this is an 11." The heartbleed attack allows an attacker to retrieve a block of memory of the server up to 64kb in response directly from the vulnerable server via sending the malicious heartbeat and there is no limit on the number of attacks that can be performed. It opens doors for the cyber criminals to extract sensitive data directly from the server's memory without leaving any traces. An attacker can manage to obtain the private encryption key for an SSL/TLS certificate and could set up a fake website that passes the security verification. An attacker could also decrypt the traffic passing between a client and a server. Schneier blogged: "the probability is close to one that every target has had its private keys extracted by multiple intelligence agencies." It turned out that flawed code was inadvertently added on New Year's Eve in 2011 and the flaw was spotted in April 2014 [126].

The Stuxnet virus that ravaged Iran's Natanz nuclear facility "was far more dangerous than the cyber-weapon that is now lodged in the public's imagination," as per cybersecurity expert Ralph Langer [90]. The exploit had a previous element that was much more complicated and "changed global military strategy in the 21st century," according to Langer. The lesser-known initial attack was designed to secretly "draw the equivalent of an electrical blueprint of the Natanz plant" to understand how the computers control the centrifuges used to enrich Uranium [111]. Only after years of undetected infiltration did the U.S. and Israel unleash the second variation to attack the centrifuges themselves and self-replicate to all sorts of computers.

The impact of the first virus was much greater. That attack provided a useful blueprint to future attackers by highlighting the royal road to infiltration of hard targets – humans working as contractors.

A recent report by the Citizen Lab [26] describes how Deep Packet Inspection (DPI) devices used by internet service providers have been misused to redirect hundreds of users in Turkey and Syria to nation-state spyware when those users attempted to download certain legitimate Windows applications. The report also describes how the DPI devices may have been used to hijack Egyptian Internet users' unencrypted internet connections en masse, and redirect the users to revenue-generating content. These misuses of DPI illustrate potential threats to human rights.

The somber reality is that at a global scale, pretty much every single industrial or military facility that uses ICS at some scale is dependent on its network of contractors. As one of the architects of the Stuxnet plan told [111]: "It turns out there is always an idiot around who doesn't think much about the thumb drive in their hand." Given that the next attackers may be terrorist organizations, civilian critical infrastructure becomes a troubling potential target. Most modern plants operate with a standardized industrial control

system, so an attacker who gets control of one industrial control system can infiltrate dozens or even hundreds more of the same breed. While governments can work hard to secure their own mission-critical facilities, the attackers can target the defense contractors by planting malware in mission-critical systems they build for the government. As the Pentagon was in the final stages of formalizing a doctrine for military operations in cyberspace, big defense contractors Lockheed Martin, Northrop Grumman, and L-3 Communications were hit by cyberattacks [38]. These attacks suggest that intruders obtained crucial information, possibly the encryption seeds for SecurID tokens, that they used in targeted intelligence-gathering missions against sensitive U.S. targets. SecurID adds an extra layer of protection to a login process by requiring users to enter a secret code number in addition to their password. The number is cryptographically generated and changes every 30 seconds. The possible attack surface is extremely broad, it doesn't have to be intrusion through the corporate network; attackers can use a disgruntled employee to plant malware.

4.4 IoT Risk

It may have only taken one click on a link that led to the download of malware strains like WannaCry [35] to set off cascading global victims of the malware. It just shows that humans will always represent the soft underbelly of corporate defenses.

The Internet-of-Things (IoT) is going to make the security risk much worse. Connected devices are proliferating at a rate IT departments and security teams can't keep up with. They are manufactured with little oversight or regulatory control, and are all WiFi and Bluetooth-enabled. We may use Amazon Echos for convenience and productivity gain. However, such devices, designed to listen and transmit information, also introduce unquantifiable risks. Recent research [8] demonstrated that the Amazon Echo is susceptible to airborne attacks. Amazon has patched the vulnerabilities, but this finding demonstrates how easily a compromised device can lead to the leak of confidential information. Attacks are coming at businesses from all channels, with IoT creating a significantly larger attack surface. Businesses are likely to face a range of consequences, from brand damage to recovery costs and loss of customers in the face of breaches. The stakes are higher than ever to secure systems and networks.

The cyber-attack that brought down much of the Internet in the US in October 2016 was caused by a new weapon called the Mirai botnet and was likely the largest of its kind in history, experts said [13]. The cause of the outage was a distributed denial of service (DDoS) attack, in which a network of computers infected with special malware, known as a "botnet", are coordinated into bombarding a server with traffic until it collapses under the strain. The victim was the servers of Dyn, a company that controls much of the internet's domain name system (DNS) infrastructure. It was hit on 21 October 2016 and remained under sustained assault for most of the day, bringing down sites including Twitter, the Guardian, Netflix, Reddit, CNN and many others in Europe and the US.

What makes the Mirai botnet interesting is, the Mirai botnet is largely made up of so-called IoT devices such as digital cameras and DVR players. Because it has so many internet-connected devices to choose from, attacks from Mirai are much larger than what most DDoS attacks could previously achieve. Dyn estimated that the attack had involved *100,000 malicious endpoints*, and the company, which is still investigating the attack, said there had been reports of an extraordinary attack strength of 1.2Tbps.

There is simply no way to rely on humans to avoid security breaches with IoT. IoT complicates matters further. Traditional solutions, such as training employees, will not mitigate the massive security challenge companies and government organizations are facing. The scope of IoT is far too complex for traditional security teams to manage with legacy solutions. There is much debate over the effectiveness of security and awareness training. It cannot be denied, however, that in the age of increased social-engineering attacks and unmanaged device usage, reliance on a human-based security strategy is questionable at best.

4.5 Critical Need: Software Assurance

Take a second look at the flawed Apple SSL code segment shown in Section 4.3.1. Is the unconditional "goto fail" a bug or malware? It could easily be an inadvertent cut and paste error. Or else, it could also be an intentionally planted line of code to enable espionage. Initially there was speculation that the Heartbleed flaw was deliberately created by government agencies to spy on citizens. Later, a developer came forward and confessed to causing the problem [25].

The real question is: *can we detect catastrophic software vulnerabilities - whether intentionally planted or not?* If we were to guard our reactors from Stuxnet, how could we have done that? There is no escape but to create the best possible technology to analyze mission-critical software to discover and confirm intentional malware or inadvertent vulnerability that could be catastrophic. But oddly enough, much of the activity that takes place under the guise of computer security is not really about solving security problems at all; it is about cleaning up the mess that security problems create. Virus scanners, firewalls, patch management, and intrusion detection systems are all means by which we make up for shortcomings in software security.

It is imperative to work software security as deeply into the development process as possible and taking advantage of the engineering lessons software practitioners have learned over the years. Two excellent books on secure programming [117, 87] advocate combining code review with static analysis tools and architectural analysis. The programming community tends to repeat the same security mistakes. Almost two decades of buffer overflow vulnerabilities serve as an excellent illustration of this point. In 1988, the Morris worm [23] made the Internet programming community aware that a buffer overflow could lead to a security breach, but as recently as 2004, buffer overflows were the number one cause of security problems cataloged by the Common Vulnerabilities and Exposures (CVE) Project [11].

Machine learning or completely automated static analysis are not adequate for software assurance. Unlike the widely studied malware in the wild, the catastrophic malware aimed at mission-critical systems is uniquely designed for a target and it can remain in a stealth mode until a trigger activates it. With uniquely designed malware, machine learning is futile. With difficult to track data and control flows [93] and state-space explosion [59], automated static analysis becomes significantly inaccurate and unscalable. While code reviews for known vulnerabilities are warranted and useful, they not an adequate solution for guarding mission-critical software from catastrophic vulnerabilities or malware that cannot be captured by signatures derived from known vulnerabilities. The complex design of catastrophic malware requires sophisticated modeling, analysis, and verification not addressed by the current automated static analysis tools [19, 10, 30] approach used for code review.

DoD Directive 3020.40 for the Defense Critical Infrastructure Program (DCIP) [14] defines Mission Assurance (MA) as "a process to ensure that assigned tasks or duties can be performed in accordance with the intended purpose or plan. It is a summation of the activities and measures taken to ensure that required capabilities and all supporting infrastructures are available to the DoD to carry out the National Military Strategy." In accordance with this directive, a principal responsibility of a commander is to assure mission execution in a timely manner. The reliance of a Mission Essential Function (MEF) on cyberspace makes cyberspace a center of gravity an adversary may exploit and, in doing so, enables that adversary to directly engage the MEF without the employment of conventional forces or weapons.

The paper *Science of Mission Assurance* [89] from the Air Force Research Laboratory (AFRL) introduces warfare in the cyber domain, identifies the weaknesses of the traditional approach to building reliable systems, and leads to an alternative approach that seeks to build secure systems. Engineering focuses traditionally on designing, developing, building, testing, and deploying complex systems that operate reliably in a permissive environment, but fail catastrophically in a contested environment. Mistaking reliability for security characterizes a generation of military, industrial, and financial systems that make little to no provision for functional vulnerability to cross-domain cyber threats. The ultimate goal of mission assurance is to develop an engineering culture that mathematically represents the specifications of a critical MEF and verifies its implementation. The paper [89] calls for a tool for reasoning on security properties and proving certain relationships among vulnerabilities and threats.

4.5.1 Software Assurance: Federal Research Programs

Software assurance and cybersecurity research and education projects are a top priority for several US Federal agencies including National Science Foundation (NSF), Department of Homeland Security (DHS), Army Research Office (ARO), Army Research Laboratory (ARL), Office of Naval Research (ONR), Naval Research Laboratory (NRL), Air Force Office of Scientific Research (AFOSR
), Air Force Research Laboratory (AFRL), Defense Advanced Research Agency (DARPA), National Security Agency (NSA), and the National Institute of Standard and Technology (NIST).

DARPA has been at the forefront of funding projects to address software assurance in the context of sophisticated malware of particular interest to national defense. DARPA research played a central role in

launching the Information Revolution. The agency developed and furthered much of the conceptual basis for the ARPANET prototypical communications network launched nearly half a century ago, and invented the digital protocols that gave birth to the Internet. The agency that brought us the Internet is working to secure it. To illustrate, we will briefly describe here goals of a few DARPA programs of particular interest to this survey topic.

Automated Program Analysis for Cybersecurity (APAC) Program: This program aimed at developing new automated program analyses capable of proving that programs have security properties of interest to the Department of Defense (DoD), and to demonstrate those analyses in the form of tools designed specifically to keep malicious code out of DoD Android-based mobile application marketplaces [5].

Vetting Commodity IT Software and Firmware (VET) Program: This program aimed at developing new techniques and tools for demonstrating the absence of backdoors and other hidden malicious functionality in the software and firmware shipped on commodity Information Technology (IT) devices. The VET program seeked to demonstrate that program analysis, coupled with updatable checklists of entries that each rule out broad classes of hidden malicious functionality, can provide a more effective proactive defense than that provided by present-day Anti-Malware products that use structural or behavior-based signatures to detect malware [41].

High Assurance Cyber Military Systems (HACMS) Program: This program aimed at creating technology for development of high-assurance software for cyber-physical systems. HACMS called for a clean-slate, formal methods based approach that enables semi-automated code synthesis from executable, formal specifications. In addition to generating code, such a synthesizer is expected to produce a machine-checkable proof that the generated code satisfies the functional specification as well as security and safety policies. [78].

Space/Time Analysis for Cybersecurity (STAC) Program: This program aims to develop new program analysis techniques and tools for identifying vulnerabilities related to the space and time resource usage behavior of algorithms, specifically, vulnerabilities to algorithmic complexity and side channel attacks. STAC seeks to enable analysts to identify algorithmic resource usage vulnerabilities in software at levels of scale and speed great enough to support a methodical search for them in the software upon which the U.S. government, military, and economy depend. [29].

DARPA programs typically involve Blue and Red teams and one White team. Blue teams develop the technology for high-assurance software. Red teams develop attacks to assess and in the process help to evolve the technology developed by the Blue teams. The White team coordinates the interactions between the Blue and Red teams and develops performance assessment measures. Over a period of 3-4 years about 100 or more highly sophisticated attacks are presented to the Blue teams in the form of on-site and off-site engagements. The progress is assessed with respect to the accuracy and scalability of techniques and tools developed by the Blue teams. Sometimes, there is also a control team that uses off-the-shelf tools for the purpose of comparison. Periodically, the tools developed by the Blue teams are delivered to the Red teams so that they understand the strengths and weaknesses of the tools and design new attacks to challenge the tools. Overall, it is a competitive environment to drive both the practical and theoretical advances in research. DARPA often seeks to develop practical tools that will not require PhDs to operate but can be used by a large work force of trained professionals.

4.6 Critical Need: Practical Tools and Cyberforce Training

It is important that the cybersecurity research leads to practical tools and education to train a cyberforce with the necessary thinking skills to use the tools effectively.

A skilled cybersecurity workforce is needed to meet the unique cybersecurity needs of critical infrastructure, enterprise, and operational technology systems and networks. In USA, the National Initiative for Cybersecurity Education (NICE) led by NIST is a partnership between government, academia, and the private sector working to energize and promote a robust network and an ecosystem of cybersecurity education,

training, and workforce development [43]. As the threats to cybersecurity and the protections implemented grow and evolve, a cybersecurity workforce must be prepared to adapt, design, develop, implement, maintain, measure, and understand all aspects of cybersecurity.

4.6.1 Human-in-the-Loop Automated Analysis

Detecting catastrophic malware in mission-critical software is an extremely complex problem. As elaborated in the paper [93] we need tools that facilitate human-in-the-loop approach. Generally, an automated malware detection tool runs in three steps: (1) a human specifies the software to be analyzed and analysis parameters, (2) the tool runs on the input and outputs a report of potential anomalies in the software, (3) an analyst goes through the report. A tool is considered sound and complete if it reports all anomalies in the software with no false positives or false negatives. However, quite often it not possible to build a sound tool. Balancing coverage vs. accuracy in an analysis tool involves an inherent trade-off: one can list only true-positives (low coverage, high accuracy) or one can output all potential anomalies (high coverage, low accuracy). Achieving high coverage and high accuracy in a fully automated tool can be impossible or incur prohibitive cost in terms of implementing the automation and/or sifting through the large number of erroneous results manually.

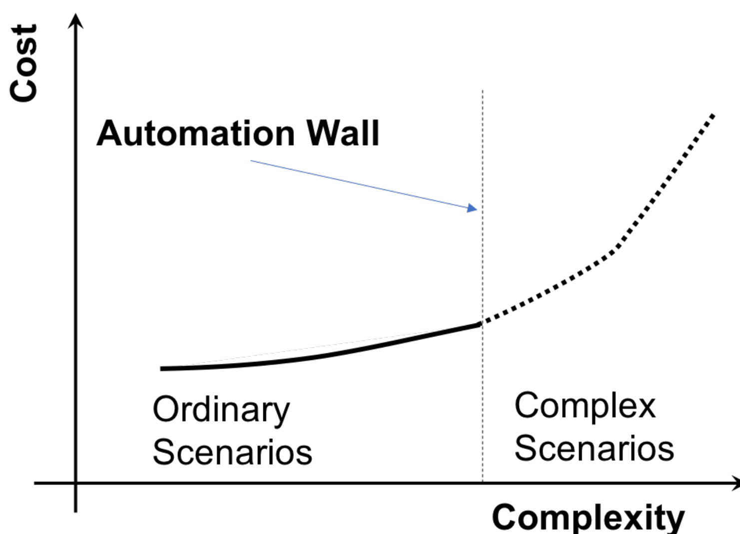


Figure 3: Analysis cost escalates beyond automation wall

To understand the need for a human-in-the-loop approach, let us classify the set of anomaly-prone scenarios into two groups: *ordinary* and *complex*. Ordinary scenarios correspond to the scenarios that are amenable to automation and do not pose extraordinary analysis challenges. On the contrary, complex scenarios are the ones that pose significant barriers to full automation. Even if automation for a complex scenario is possible, it may well be infeasible due to economics of time and effort. As summarized in Figure 3, static analysis tools hit an *automation wall* and the cost for resolving complex scenarios escalates beyond the automation wall.

The human-in-the-loop analysis paper [93] advocates *Amplified Reasoning Technique* (ART) for analyzing software for complex malware. The ART philosophy is: Instead of resolving complex anomalies as definitive Yes/No answers through a fully automated tool, bring the human in a man-machine loop and use the tool to amplify human reasoning to resolve such anomalies faster and efficiently, so that it can be scaled to large software.

Figure 4 brings out the difference between the traditional approach to automation vs. the ART. In the traditional automation, the role of human is to sift through the false positives and unresolved cases generated from the automation run, and is segregated from the role played by the machine. Whereas, the ART puts the human and machine in an interactive loop.

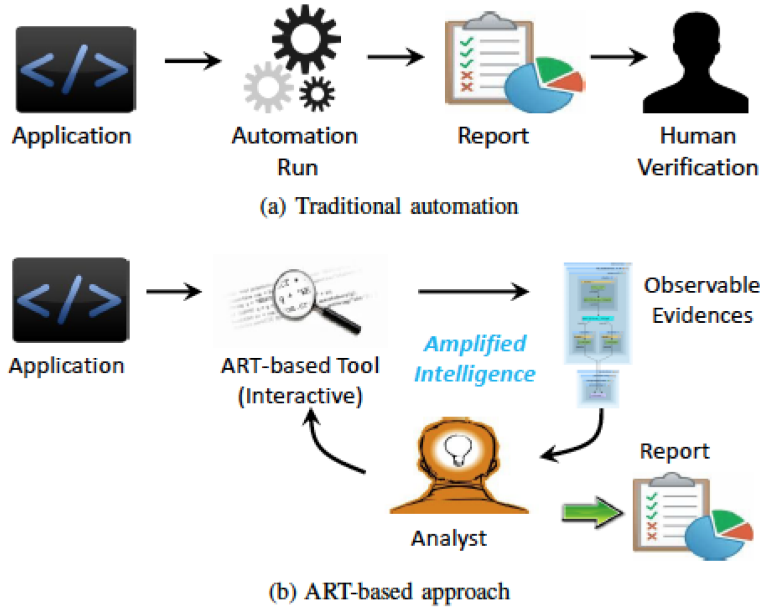


Figure 4: Traditional automated malware analysis vs. ART

Analyzing software for a zero-day mission-critical flaw (a kind of flaw that has never been seen before) is like looking for a needle in the haystack, but without knowing what the needle looks like. As we shall discuss later, the first step in detecting such a flaw requires us to develop plausible flaw hypotheses and then gather evidence to prove or refute each hypothesis. The open-ended task cannot be performed automatically. Developing plausible a hypothesis requires human intelligence and gathering knowledge about the software. Without a human-in-the-loop tool this task of gathering knowledge from software can be extremely time consuming and prone to human errors. Thus, human-in-the-loop tools are essential for detecting zero-day mission-critical software flaws.

4.6.2 Tools for Transparent Verification

Software verification is especially important in the context of CPS for critical applications, where failures have had catastrophic consequences. Verifying software is an important but daunting task with challenges of accuracy, scalability, and practicality. Formal verification of large software has been an elusive target, plagued with the problem of scalability [59, 53, 124]. Two fundamental limitations are: (1) a completely automated and accurate analysis required for formal verification encounters NP hard problems [61, 120, 110], and (2) formal verification methods work as automated black boxes with very little support for cross-checking [127, 75, 53].

The second limitation leads to issues that leave several practical needs unaddressed. Although the formal verification works as an automated black box, it requires an inordinate amount of preprocessing effort, involving a transformation from the software to the formal specification that can be checked automatically using a model checker or a SAT solver. This transformation is not automatic, it requires domain knowledge of the particular formal method and a lot of cumbersome human effort.

Besides the preprocessing, another serious issue is the lack of supporting evidence to be able to understand and use the results of formal verification. Without the evidence, it is not possible to use formal verification as a certification apparatus, or to integrate formal methods in a development environment. As we shall exemplify, it is quite hard for the user to know that the verification result is wrong without supporting evidence. We will present an empirical study to elaborate the notion of evidence and its importance in practice.

Leaning on visionary papers [73, 55] by Turing Award recipients, the paper [92] explores the question: “What advances in formal methods would it take to meet the practical needs?” The paper [92] presents

empirical study that provides deep insights into some of the state of the art formal verification methods. The study uses the Linux Driver Verification tool (LDV) [53] which has been the top Linux device driver verification tool in the software verification competition (SV-COMP) [51]. The study includes three versions of the Linux operating system with altogether 37 MLOC and 66,609 verification instances. Each instance involves verifying that a `Lock` is followed by `Unlock` on all feasible execution paths. Running LDV on these Linux versions yields the result that pairing is correct for 43,766 (65.7)% of `Lock` instances. LDV is inconclusive on 22,843 instances, i.e. either the tool crashes or it times out. LDV does not find any instance with incorrect pairing. LDV does not provide evidence to support its results except for the instances where the verification reveals a bug.

5 Difficulties of Detecting CPM

It has been hard to be proactive to stop adversaries from mounting new CPM attacks. The core difficulty of detecting CPM lies in the open-ended possibilities for malware triggers and payloads, the subtle boundary between "malicious" and "legitimate," the obfuscation and other ways to impede program comprehension that is necessary to detect malware. Fundamentally, the challenges of detecting sophisticated CPM stem from the complexities inherent in the software itself.

It is almost impossible to improve software security merely by improving quality assurance. In practice, most software quality efforts are geared toward testing program functionality. Most software testing is aimed at comparing the implementation to the requirements, and this approach is inadequate for finding security problems. Imagine testing a piece of software by running down the list of requirements and making sure the implementation fulfills each one. It will miss many security problems because security problems are often not violations of the requirements. Instead, security problems are frequently "unintended functionality" that cause the program to be insecure.

To begin with, the key difficulty is to first hypothesize possibilities for malware. Software theorists are after executable specifications, automated model checking, and theorem proving in order to verify software. Software practitioners are after perfecting the art of penetration testing. How can the theorists or the practitioners effectively apply their knowledge to CPM? If we were to use today's software verification tools based on Formal Methods (FMs) to verify the GPS software, how do we decide what to prove?

5.1 GPS Malware: An illustrative example

Detecting CPM requires the knowledge of *malware triggers* and *malware payloads*. Malware payload refers to the part of the software that, when executed, actually causes the damage. Malware trigger refers to the conditional part of the software that, when the condition is met, the execution follows a control flow path with the malware payload. Thus, malware is only activated when properly triggered. Code paths implementing malicious behaviors are executed only when certain trigger conditions are met.

Imagine a GPS device that works accurately, except it malfunctions in Afghanistan on full moon days. No matter how exhaustive the reliability testing is in their Kansas City factory, the GPS manufacturer will not catch mission-critical malware that malfunctions only in a certain geographical region. Figure 5 shows a code snippet for the malware. The conditional part of the software that enacts the trigger is shown in a box and the geographic region it specifies is shown to the right. The malware payload consists of the `location.setlongitude()` and `location.setlatitude()` calls that result in malicious modification of the longitude and latitude information when the GPS device operates in the particular geographic region. We shall use this malware example to illustrate why it is hard to detect CPM.

5.2 Why it is difficult to detect CPM

In contrast to the limited triggers for traditional computer malware, the sensor inputs lead to open-ended possibilities for CPM triggers. The traditional malware is limited by typical inputs to a computer such as the keyboard, file, or mouse etc. Moreover, inputs are explicit and can be associated with a short list of program artifacts such as `read` or `get` statements. Thus, traditional computer malware can be detected by auditing such statements. For example, the size of the input can be checked to avoid the buffer overflow

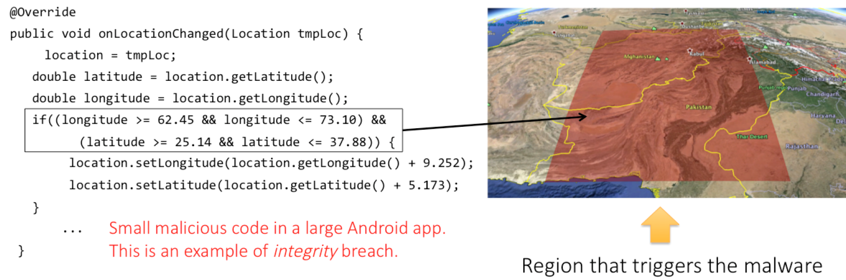


Figure 5: GPS Malware with obscure trigger

attack. CPM detection becomes tricky because of sensory inputs, as the physical environment itself can be the input, not just the user input. For example, the geographic region is the input for the GPS malware.

Currently, trigger-based malware analysis is often performed in a tedious, manual fashion. The paper [56] on automatically identifying trigger-based malware states that there is no previous work on automating trigger-based malware analysis. It discusses integration of techniques from formal verification, symbolic execution, binary analysis, and whole-system emulation and dynamic instrumentation to enable automatic identification and analysis of trigger-based behaviors in malware. Even when complete automatic analysis is not possible, they claim that their system still provides valuable information about potential trigger-based code paths which a human would otherwise have to discover manually.

Identifying trigger-based behaviors in malware is an extremely challenging task. The paper [56] considers trigger-based denial-of-service attacks. Attackers are free to make code arbitrarily hard to analyze. This follows from the fact that, at a high level, deciding whether a piece of code contains trigger-based behavior is undecidable, e.g., the trigger condition could be anything that halts the program. Thus, a tool that uncovers all trigger-based behavior all the time reduces to the *halting problem*.

In differentiating CPM from traditional malware, the difference really comes from the open-ended possibilities for malware triggers and payload. The open-ended possibilities are tied to the wide spectrum of sensor inputs and the almost limitless application-specific possibilities for designing the payloads. In the GPS malware, the payload is designed to corrupt the location information given to the user. In Stuxnet, the payload is designed to corrupt the behavior of the algorithm that controls the centrifuges. Both fall under the integrity breach according to CIA triad. However, the breaches are so application-specific that knowing that it is an integrity breach and the understanding of the breach in one application, hardly helps in hypothesizing the malware for the other application. In other words, the difficulty of detecting CPM is really the difficulty of open-ended possibilities.

The subtle boundary between "malicious" and "legitimate," is another difficulty that is exacerbated by the open-ended possibilities. Consider the GPS malware example. The modifications of latitude and longitude is per se not as malicious as GPS software that includes legitimate modifications. Significant domain knowledge about the application is often required to draw the boundary between "malicious" and "legitimate." In the case of the GPS malware the trigger region does hint at the possibility of malware.

6 Software Assurance: Fundamental Challenges

Fundamentally, the challenges of detecting sophisticated CPM stem from the complexities inherent in the software itself. Add to a simple calculator with arithmetic operations two features: *conditional computation* and *store and recall* and the calculator becomes a powerful Turing complete computer [109]. The first feature leads to the *control flow* (CF), and the second feature to the *data flow* (DF). The fundamental challenges of analyzing software stem from data and control flow.

6.1 Memory Leak: An illustrative example

This example is taken from XINU [66], a small operating system. It brings out the challenges of analyzing software to verify its safety and security properties. In this example, the problem is to verify that memory allocation is followed by deallocation on all feasible *control flow* (CF) paths. The allocation and deallocation system calls are respectively `getbuf` and `freebuf`.

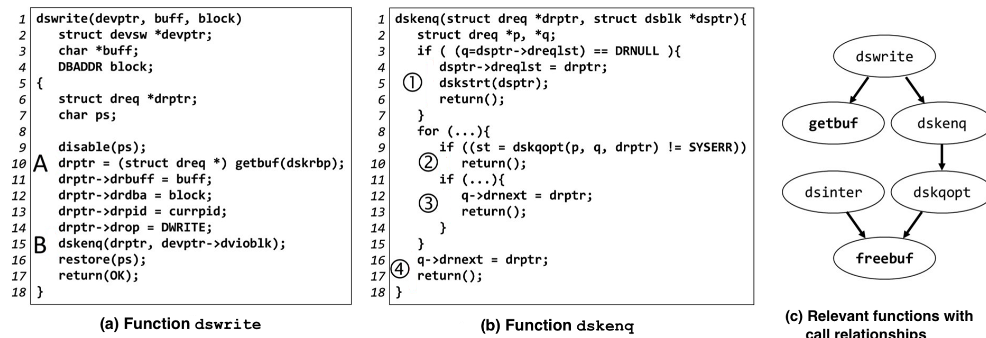


Figure 6: The importance of evidence to reason about the possibility of a memory leak in the function `dswrite`

The starting point is the `dswrite` function shown in Figure 6(a). Function `dswrite` calls `getbuf` but it does not call `freebuf`. The verification problem is to match the `getbuf` call in `dswrite` with the corresponding `freebuf` call(s) which would be in other functions.

As seen from the Figure 6(a), `dswrite` passes the allocated memory pointer `drptr` to function `dskenq`. As shown in Figure 6(b), `dskenq` has four CF paths. On path 1: `drptr` is passed to function `dskstrt` which does not free the allocated memory. But, we cannot conclude that it is a memory leak because on the same path `drptr` is assigned to `dsptr->dreqlst` where `dsptr` (passed as a parameter to `dskenq` from `dswrite`) points to a global linked list. On path 2: `drptr` is passed to function `dskqopt`. On paths 3 and 4: `drptr` is assigned to `q->drnext` and earlier `q` is set to points to `dsptr->dreqlst`. Thus, on three paths (1, 3 and 4) the allocated memory is not freed but the pointer to the allocated memory is inserted in a global linked list. The `dskenq` code snippet in Figure 6 is incomplete; only the relevant parts are shown.

Since the pointer to the allocated memory gets passed as a parameter to other functions, the call chains must be tracked. Moreover, one path in `dswrite` multiplies into 4 paths in `dskenq` and the path proliferation continues through functions down the call chain. Analyzing the call chains and the proliferation of paths is tedious and challenging.

Even more challenging is to analyze the part where the pointer to the allocated memory is assigned to a global linked list. Since the memory pointer is accessible through a global variable, any function could access that pointer and free the memory. Do we then examine and verify all functions? It would be a huge verification challenge.

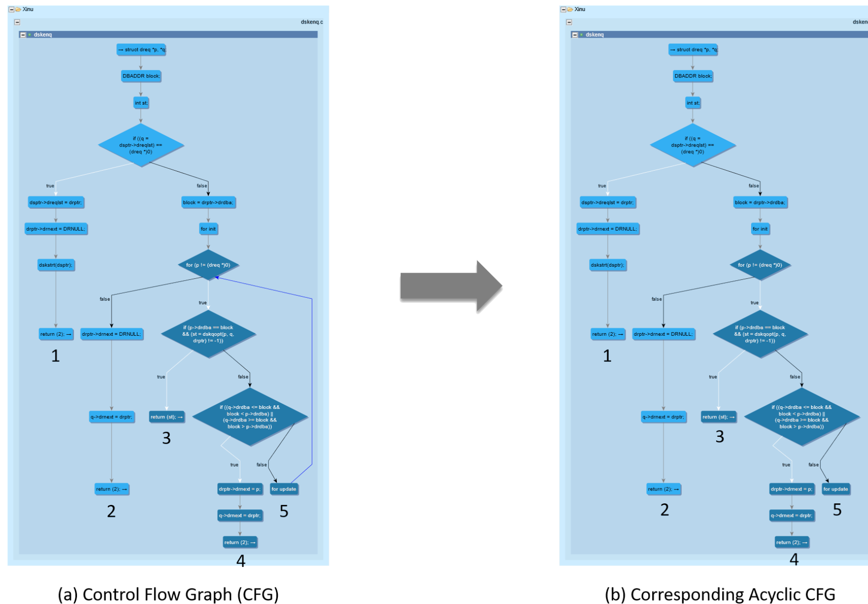
The reality is, only `dswrite`, `dskenq`, `dskqopt`, `dsinter` are the relevant functions to be analyzed to reason about this memory leak example. The verification can be completed by analyzing just these functions. The call relationships among these functions are shown in Figure 6(c). Starting with the allocation in `dswrite`, all the CF paths go through `dskenq`, and `dskqopt`. One of the two things happen on these CF paths: (a) the `getbuf` calls in `dskqopt` deallocate memory, or (b) the pointer to the allocated memory is assigned to a global linked list. The function `dsinter`, an interrupt-driven function, gets called asynchronously. It goes through the linked list and deallocates the memory for each pointer in the list until the list becomes empty. We can thus verify that there is no memory leak. We will use this example to exemplify each fundamental challenge listed in the next subsection. The challenge is how to discover the relevant functions efficiently. Without such discovery, we are faced with examining all functions.

6.2 Explosion of Execution Behaviors

Verifying a safety or security property usually requires analysis of all paths. That can make software assurance computationally intractable. An `IF` statement creates two paths where as a function call expands a path

into m paths where m is the number of paths in the function. With b non-nested conditions, the number of paths is 2^b.

Let us illustrate how the computational intractability is exacerbated by program loops. The *control flow graph* (CFG) for the function `dskenq` shows one loop (Figure 7(a)). The *loop header*, the entrance to the loop, is the first reachable node shaded dark blue in the figure. The loop-back edge is colored blue and the true and false branches are shown by white and black edges respectively. The loop has three **termination nodes**, defined as the nodes that have a successor that is outside the loop body. The loop header is one of the termination nodes and it is referred to as the *normal termination node*. Other termination nodes due to `break` or `return` are referred to as *exceptional termination nodes*. The loop has two such termination nodes due to `return`. The paths from those termination nodes lead to the `return` statements marked 3 and 4 in Figure 7(a).



(a) Control Flow Graph (CFG)

(b) Corresponding Acyclic CFG

Figure 7: Counting CF paths with CFG and associated acyclic CFG

The loop behaviors can be computed as iterations of *base behaviors* of a loop. Each base behavior corresponds to one iteration of a loop. The base behavior is the sequence of program statements executed during one iteration of the loop. The base behaviors for loops are partitioned into: (a) *normal base behaviors* (B^N) - the behaviors along the paths that terminate at the normal termination point, and (b) *exceptional base behaviors* (B^E) - the behaviors along the paths that terminate at the exceptional termination points.

An **iterative behavior** is a sequence of base behaviors of length i, where i is a positive integer representing the number of iterations of a loop. For $i > 0$ iterations, n normal base behaviors, e exceptional base behaviors, the number of iterative behaviors is: $(n^i + e \times n^{i-1})$ with (n^i) iterative behaviors that do not end with an exceptional base behavior, and $(e \times n^{i-1})$ iterative behaviors with $i-1$ iterations of normal base behaviors followed by a final iteration of an exceptional base behavior. Note that the exceptional behavior can only be at the end because the loop is terminated after the exceptional behavior.

The base behaviors can be discerned by breaking the loop-back edges to create an acyclic graph. Figure 7(a) shows the CFG with a loop and Figure 7(b) shows the corresponding acyclic graph. These graphs are generated by using the path counter tool built using Atlas [71]. The layouts are different but the correspondence between the CFG and its associated acyclic graphs are shown by numbering the corresponding `return` nodes. The node numbered 5 is interesting. It is the tail of the loop-back edge in Figure 7(a). It is a leaf node in the acyclic graph representing the normal base behavior. The normal base behavior is the sequence of program statements from the loop header to the node numbered 5. The exceptional base behaviors are the sequences of program statements from the loop header to the nodes numbered 3 and 4

respectively. Thus, we have one normal base behaviors B_1^N and two exceptional base behaviors B_1^E and B_2^E.

The maximum number of normal base behaviors for a loop is $n = 2^b$, where b is the number of branch nodes in the loop body. Then, a successive exponentiation is due to iterations of a loop. Suppose the acyclic graph of a loop has n normal base behaviors and e exceptional base behaviors, then for i iterations, it has $(n^i + e \times n^{i-1})$ iterative behaviors. The double exponentiation leads to utter intractability if a brute force approach were to be applied to reason about software safety and security problems that require analysis behaviors on all paths. The number of behaviors is bigger than the number of atoms in the universe if we have a loop with just 5 non-nested IF conditions and 50 iterations.

The exponentiality of paths coupled with explosion due to iterative behaviors are the root problems that go by various names in software analysis and verification literature. For example, Clarke et al. [63] discuss the state explosion problem for model checking. Model checking is an automatic verification technique for concurrent systems that are finite state or have finite state abstractions. Model checking is a collection of automatic techniques for verifying finite-state concurrent systems. This framework was developed independently in the early 1980's by Clarke and Emerson [62] and by Queille and Sifakis [107, 108]. It has been used successfully to verify computer hardware, and it is beginning to be used to verify computer software as well. As the number of state variables in the system increases, the size of the system state space grows exponentially. This is called the *state explosion problem.* Much of the research in model checking over the past 40 years has involved developing techniques for dealing with this problem. The behavior associated with each path creates a set of states. As the number of paths grows exponential, so does the totality of states associated with those paths. The iterative behaviors, as discussed here, cause a successive explosion. Thus the state explosion problem is rooted in explosion of behaviors.

6.3 Computational Intractability of Path Feasibility

Given a description of a set of control flow paths through a procedure, feasible path analysis (FPA) determines if there are appropriate input values that would cause execution to flow down some path in the collection [82]. If no input values can cause the program to be executed along a path, we say that the path is *infeasible* or *non-executable.* In the context of software testing, feasibility analysis plays an important role in identifying testing requirements which are infeasible. In software assurance, the analysis may find a path with a safety or security vulnerability. Declaring it to be an actual vulnerability becomes a false positive if the path is not feasible.

FPA is central to most applications of program analysis. But, because this problem is formally unsolvable, syntactic-based approximations are used in its place. For example, the dead-code analysis problem is to determine if there are any input values which cause execution to reach a specified program point. The approximation determines whether there is a control flow path from the start of the program to the program point of interest . This syntactic approximation is efficiently computable and conservative: if there is no such path the program point is clearly unreachable, but if there is such a path, the analysis is inconclusive, and the code is assumed to be live. Such conservative analysis too often yields unsatisfactory results because the approximation is too weak.

Feasible path analysis requires both symbolic analysis [91] and theorem proving [50]. Symbolic analysis relates expressions occurring at different program points and theorem proving determines the validity of logical relations between expressions. A popular approach is to use *constraint solvers* - firstly extract a set of constraints from the given path, and then solve the constraints [129]. Given a path, we can obtain a set of constraints called *path predicates* or *path governing conditions.* After the path conditions are obtained, we need to decide whether they are satisfiable or not. For satisfiable conditions, we often have to find the variables' values to satisfy them. The path is feasible if and only if these conditions are satisfiable.

A Constraint Satisfaction Problem (CSP) [99] consists of a set of variables, each of which can take values from some domain. In addition, there are some constraints defined on the variables. Solving a CSP means finding a value for each variable, such that all the constraints hold. Obviously, CSP represents a very general class of problems. Special cases of interest in software analysis are: *linear inequalities as constraints* and *Boolean satisfiability.*

A Boolean variable can only be assigned some truth value (`TRUE` or `FALSE`). A Boolean formula is constructed from a set of Boolean variables using the logical operators `AND` (i.e., conjunction), `OR` (i.e., disjunction), etc. A literal is a variable or its negation. The disjunction of a set of literals is a clause. A Boolean formula can

be transformed into a conjunction of clauses. If we can assign a truth value to each variable such that the whole formula evaluates to `TRUE`, then the formula is said to be satisfiable. If the formula is in conjunctive normal form (CNF), i.e., it is a conjunction of clauses, the problem is well-known as SAT. This is the first NP-hard problem [81].

6.4 Difficult to Analyze Programming Constructs

As noted earlier, two simple but powerful programming constructs: *conditional computation* and *store and recall* are at the heart of a Turing-complete computation model. Modern programming languages have added new programming constructs to enable paradigms such as: structured programming, object-oriented programming, collections (e.g., structures, arrays, etc.), and dynamic binding.

These new programming constructs have their benefits, but they create new complexity beyond the two fundamental challenges we have discussed. The complexity arises from inter-dependencies between data and control flows created by these programming constructs. We view it as *indirect* and *invisible* flows that complicate program comprehension and also make automated program analysis difficult.

6.4.1 Indirect Control Flow

Control Flow (CF) is about the order in which program statements are executed. The ability to modify control flow enables the construction of program loops and provides the ability to create multiple execution behaviors. CF artifacts such as the `IF` statement or a function call modify the linear order in which program statements are executed. The CF artifacts provide the capability to create different execution behaviors in software.

We use the term direct control flow to refer to flow created by program constructs `IF` and `function call` that directly specified that flow, i.e. no computation is needed to discern the flow. This is not the case with *indirect control*. Take the case of dynamic binding in C as exemplified by a device driver call in XINU:

```
SYSCALL write(descrp, buff, count)
        int descrp, count;
        char *buff;
        {
                struct devsw *devptr;

                if (isbaddev(descrp))
                        return(SYSERR);
                devptr = &devtab[descrp];
                return((*devptr->dvwrite)(devptr,buff,count));
        }
```

The function `write` calls a function `(*devptr->dvwrite)(devptr,buff,count)` using a function pointer. To compute the control flow at the callsite, we must identify the called function. However, we need to compute data flow to do so. We have a complex dependency: to compute the control flow we need to compute the data flow. In turn, to compute the data flow we always need control flow. Moreover, depending on the path by which arrive to the `write` call, the function called through the function pointer could be different. In essence, the function pointer call creates a branch point at the callsite and adds further complexity to the fundamental challenge of explosion of behaviors.

We use the term *indirect control flow* to refer to function calls that require computing data flow to identify the called function.

The object-oriented languages employ *type hierarchy* for the indirect control flow. For example, to identify the function called by `O.f(x)`. A data flow computation is needed to determine the *type* of the object `O`. The called `f` is the one that is implemented for the determined *type*.

Another variety of indirect control flow stems from program constructs for handling exceptions. In Java, the `try` block contains set of statements where an exception can occur. A `try` block is always followed by a `catch` block or a `finally` block. The `catch` block handles the exception that occurs in the associated `try` block.

There are also specialized program constructs such as Android *intents*. An intent is a specialized data structure that is passed to signal dispatching logic to change an application's control flow from executing one component to another within the Android framework. Depending on the intent, apps or the OS might be listening for it and will react accordingly. Think of it as a blast email to a bunch of friends, in which you tell your friend John to do something, or to friends who can do X (intent filters), to do X. The other folks will ignore the email, but John (or friends who can do X) will react to it. To listen for a broadcast intent (like the phone ringing, or an SMS is received), broadcast receiver is implemented, to which the intent will be passed [1, 2].

The indirect control flow is used rampantly in object-oriented languages. The Object class, in the java.lang package, sits at the top of the class hierarchy tree. Everyclass is a descendant, direct or indirect, of the Object class. Every class you use or write inherits the instance methods of Object. Accurate static analysis is often unnecesarily complex as a result. In contrast, the C developers employ function pointers only when their use makes good sense. We did a quick sampling of Linux and XINU builds using Atlas [71] queries to find call sites that use function pointers. Of the total $2,325,293$ call sites in Linux $53,741$ (2.3%) use function pointers. Of the total $1,007$ call sites in XINU, 14 (1.4%) use function pointers.

A Linux bug rooted in odd use of function pointers: We present an example of a complex bug in a Linux device driver involving an odd use of function pointers. The bug is reported in our paper [92]. It shows a feasible path on which `Lock` is not followed by `Unlock`. A function `f1` is called using a function pointer. Oddly, it has dual use: serves as `Lock` or `Unlock` depending on whether its boolean parameter is `TRUE` or `FALSE`. The function has two control flow paths with `Lock` on one path and `Unlock` on the other path. The function is called twice on most paths except on one path which is the bug. This function with its dual use and its function pointer calls make it unnecessarily complex.

The lock and unlock are on disjoint paths in the function `drxk_gate_crtl` (`f1`) and if $C =$ `true`, the lock occurs, otherwise, the unlock occurs. The lock and unlock can match if `f1` is called twice, first with $C =$ `true` and then with $C =$ `false`. An Atlas query shows that `f1` is not called directly anywhere. Thus, it is either dead code or `f1` is called using a function pointer.

Resolving indirect control due to function pointers, we find the scenario shown in Figure 8. The function `tuner_attach_tda18271` (`f2`) calls the function `f1` via function pointer. `demo_attach_drxk` sets the function pointer to `f1`, the pointer is communicated by parameter passing to `dvb_input_attach`, then to `f2`.

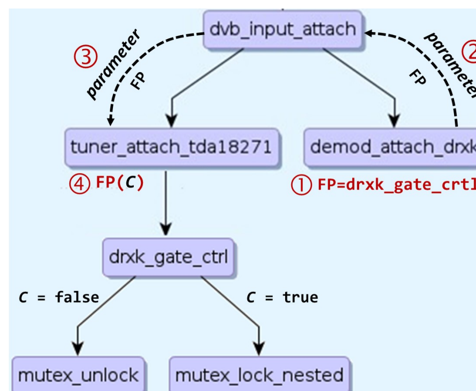


Figure 8: A Linux bug in `drxk_gate_crtl` with odd use of function pointers

Recall that `f1` must be called twice. The function `f2` has a path on which there is a return before the second call to `f1` and thus it is a bug.

6.4.2 Indirect Data Flow

Data flow is about the *store and recall* in a program. The data flow artifacts include assignment statements, function parameters, function returns, local variables, and global variables, object or structure fields, and pointers to objects or structures. The ability to flow data enables the construction of multi-part programs

where different parts can share their computations. One part (control block or function) can compute and store the result and another part of the program can recall the stored result.

Direct data flow is through assignment of variables. Data flows from variable `y` to variable `x` through assignment `x=y`. The direct data flow is directly computed by tracking assignments. The data can flow through multiple assignments, but that by itself is not a major difficulty and the Def-Use (DU) Use-Def (UD) chains are used respectively to track the *forward* and *backward* data flow [46].

Indirect data flow is the data flow through a function or a field of a structure. The *indirect data flow through a function* can be further divided into three categories: (a) parameter assignment, (b) function return, and (c) assignment to a global variable. The analysis difficulty increases from (a) to (c). In (a) the data flows into a specified function `f`. In (b), the data returned by the function `f` can flow any of its callers. In (c), the data assigned to a global variable by a function `f` can be accessed by any other function.

Indirect data flow through an object or structure field creates the complexity of *backward flow* as exemplified here:

```
1.  Lock(x);
2.  O1.a = O2;
3.  O2.b = x;
4.  y = O1.a.b;
5.  Unlock(y);
```

The assigned `x` in Line 3 flows back through the assignment in Line 2 and thus `Unlock(y)` is actually `Unlock(x)`.

6.4.3 Invisible Control Flow

Unlike *direct* or *indirect*, the invisible control flow change is not specified by any visible program artifact. Instead, an interrupt causes the change. In case of *indirect*, the callee is not directly specified but it can be computed. In case of *invisible*, neither the callee nor the program point (a statement in the program) for control change is specified. The added uncertainty of the program point makes analysis of invisible control flow even more difficult. The invisible flow is illustrated by our earlier memory leak example (Figure 6):

1. Memory is allocated by invoking `getbuf` inside the function `dswrite`. The memory is allocated for a structure of type `dreq`. The `drptr` pointer to the allocated memory is passed to other functions and eventually it is inserted in a globally shared linked list with the code `dsptr->dreqlst = drptr`.

2. The function `dsinter` gets `drptr`, and deallocates memory with the code `drptr = dsptr->dreqlst` followed by `freebuf(drptr)`.

The function `dsinter` is interrupt-driven. It is not linked by a call chain to `dswrite` which allocates memory. The `drptr` pointer to the allocated memory is communicated through a global linked list `dsptr->dreqlst`.

In invisible control, interacting program parts are not linked by control flow. They act asynchronously and communicate through a shared object.

6.4.4 Invisible Data Flow

Invisible data flow occurs when an individual variable or a pointer becomes part of a collection, looses its individual identity and cannot be tracked individually.

The invisible flow is illustrated by our earlier memory leak example (Figure 6). In `dsptr->dreqlst = drptr`, the `drptr` pointer to the allocated memory becomes part of a linked list. In fact, this linked list contains pointers to allocated memory in different functions, not just `dswrite`.

Pointer analysis attempts to determine the set of objects to which a pointer can point (called the points-to set of the pointer). Unfortunately, these analysis are necessarily approximate (since a perfectly precise static analysis amounts to solving the halting problem). Pointer analyses have difficulty analyzing invisible data flow precisely. In order to determine points-to sets, a pointer analysis must be able to name a program's objects. In invisible flow instances, programs can allocate an unbounded number of objects; but in order to terminate, a pointer analysis can only use a finite set of names.

7 Research Directions

We shall discuss new research directions in terms of: (a) threat modeling to hypothesize CPM, (b) analyzing software to gather evidence based on a CPM hypothesis, and (c) verifying to prove or refute a hypothesis based on gathered evidence.

7.1 Threat Modeling

Security threat modeling is often studied and applied for specifying security requirements during application development [117, 87]. There is however paucity of established techniques and tools for threat modeling and analysis [102]. Ongoing work at the Open Web Application Security Project (OWASP) organization includes several resources for threat modeling [44].

A different view of threat modeling is pertinent in the context of catastrophic malware that has never been seen before. Searching for such malware is like looking for a needle in the haystack not knowing what the needle looks like. The OWASP [44] notes that threat modeling is a complex endeavor that involves drawing trust boundaries. How does one draw trust boundaries in software? Drawing boundaries with respect to known exploits is doable, but doing so with unknown exploits is uncharted territory requiring new research. Threat modeling is important for the following reasons.

- Without the threat model as a powerful abstraction, we are left to deal with an endless variety of problems. For example, without the abstraction of variables and the linear system of equations, there is an endless variety of constraint-satisfaction problems. Similarly, without effective modeling of CPM, we have an endless variety of physical systems as well as their varied malfunctions.

- Threat modeling with appropriate rigor is a crucial prerequisite for efficient analysis and verification of CPM. A powerful abstraction is necessary to avoid ad-hoc and inefficient solutions. For example, the abstraction of linear system of equations has enabled the powerful Gauss and Jacobi methods that scale to extremely large constraint satisfaction problems encountered in science and engineering. Similarly, modeling is a necessity to design efficient and accurate algorithms to analyze and verify CPM.

In detecting novel CPM, the first challenge for a human analyst is the open-ended search for plausible hypotheses of malicious behavior. The space of hypotheses can be arbitrarily large for a given app. With no prior knowledge of an app's malicious behavior, the analyst has to rely on a general model such as the CIA model [69] to systematically explore hypotheses for malicious behaviors. As examples from Android apps, an analyst may consider hypotheses for confidentiality leaks (e.g., GPS location is leaked to an adversary), integrity breaches (e.g., incorrect GPS location is displayed in some geographic locations), or denial of service (e.g., malware runs loops to drain the device battery) attacks enabled by unscrupulous use of Android APIs. However, this can still be prohibitively expensive, as there may be numerous potential CIA hypotheses to explore for a given app.

Hypothesizing a vulnerability amounts to specializing the CIA model by coming up with specific events and rules surrounding the events to specify a breach. To precisely explore only the relevant CIA hypotheses, it is critically important for the analyst to visualize and understand the data and control flow interactions between the app and the Android components it uses. For example, knowing that an app interacts with the hardware and messaging APIs via data flow leads to the hypothesis that it may leak camera images by sending them as attachments with messages. Coming up with a good CPM hypothesis is a highly creative activity requiring human intelligence at its best. Based on our experience of hypothesizing malware in DARPA programs, we have found the following research directions particularly useful:

Automated Exploration: The purpose is to identify and characterize relevant program artifacts. It requires research to determine what artifacts could be relevant. The relevant program artifacts are characterized with respect to a class of malware.

Automated Filters: The purpose is to enable the analyst to quickly sift through relevant program artifacts to identify targets for formulating malware hypothesis. The analyst starts developing a hypothesis by experimenting with various filters.

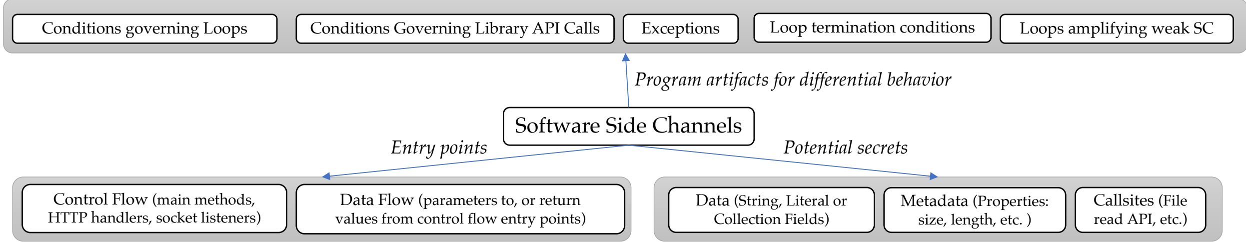


Figure 9: Three variability dimensions of SSCVs

7.2 DARPA Research: Threat Modeling

Our work on DARPA APAC and STAC projects is summarized here to exemplify the threat modeling research directions. The papers [71, 85, 72, 112, 113, 84, 49] describe our research related to threat modeling and the relevant toolboxes we have developed. These papers include research on interactive visualization and querying which analysts need, in order to understand complex software and gather relevant information during threat modeling.

7.2.1 Automated Exploration

Android Vulnerabilities: The gathered information includes the data, control and exceptional flows within the app, the permissions granted to the app and the APIs used to exercise them, and program artifacts that use the Android resource files. The pre-computed information also includes the call and data flow interactions of the app with various Android *subsystems*. Subsystems are logical groupings of Android APIs according to the functionality they provide such as networking, storage device access, address book, and others. The interactions provide information about how the app interacts with Android components.

Side Channel Vulnerabilities: Software side-channel vulnerabilities (SSCVs) allow an attacker to gather secrets by observing the differential in the time or space required for executing the program for different inputs. The possibilities are open-ended for the ways various program artifacts may be used to create side channels. Attackers exploit SSCVs by presenting a set of inputs and observing the space or time behaviors to construe the secret. The input could vary from HTTP requests to multiple log-in attempts depending on the application. The observable differential behaviors could be execution time, memory space, network traffic, or some output patterns of application-specific significance.

The paper [113] describes a three dimensional variability spectrum shown in Figure 9 corresponding to fundamental SSCV attributes: entry points, potential secrets, and programming constructs or artifacts causing differential behavior. Adversaries use *entry points* to provide inputs to induce differential behaviors, *secret types* are the broad categories of secrets that adversaries target, and *observables* are the space or time program execution behaviors produced by control flow constructs in the code such as loops, branches and exceptions. The relevant artifacts are gathered by automated exploration. It is important to characterize relevant program artifacts and their specific attributes that define how the artifacts relate to the secret, how they create observable space/time behaviors, or how they create differential behaviors.

Algorithmic Complexity Vulnerabilities: The *algorithmic complexity vulnerabilities* (ACVs) are about runtime space or time consumption of programs. Adversaries can exploit ACVs to mount denial of service attacks. For example, the denial of service commonly known as the "billion laughs attack" or an XML bomb, is caused by an ACV in the application that creates a string of 10^9 concatenated "`lol`" strings requiring approximately 3 gigabytes of memory [68] when parsing a specially crafted input file less than a kilobyte. A recent study has characterized a class of ACVs in the Java library [74]. Similar to the XML bomb, it is a class of ACVs associated with the serialization and deserialization APIs.

The program artifacts that usually lead to ACVs include loops, recursion, or resource-intensive library APIs. Moreover, ACVs typically result from complex loop termination logic. An important part of our research has been to decipher relevant loop characteristics by studying publicly known examples of ACVs

and the ACV challenges posed by DARPA. It is important to characterize loops in the context of program artifacts that connect a loop to the rest of the program. For example, since ACVs are triggered by attacker's input, it is important to characterize whether the termination of a loop can be controlled by user input. We developed loop abstractions capture and represent the essentials of loops and the connecting parts of the program that affect loop behaviors. One abstraction is to capture the loop termination behavior based on the data flow to the loop termination conditions.

The gathered information includes: (a) a catalog of all loops in the program along with characteristic attribute vector for each loop, (b) the uses of resource-intensive library APIs, (c) loop call graphs to bring out loops nested over multiple functions, (d) a catalog of branch nodes including branches that govern the paths containing loops.

7.2.2 Automated Filters

The paper [49] describes automated filters to isolate complex loops with high likelihood of ACVs. The analyst can use filters to select loops matching a combination of the loop characteristics from the loop catalog. The framework currently supports the creation of custom filters by adding constraints on String, primitive and boolean properties. An example of a boolean property is monotonicity – a loop is either monotonic or not; and the two possible constraints based on this property would be "monotonic: true" and "monotonic: false". The nesting depth of a loop is an example of a primitive (integer) property. For example, the constraint "nesting-depth greater than 4" selects all loops having nesting depth of 5 or above within the method. A filter consists of a conjunction of constraints, i.e., a filter consisting of the above two constraints would select monotonic loops with nesting depth over 4. The filtering framework also allows analysts to fork a filter, i.e., create a new filter that includes a subset of the constraints added to an existing filter. This is useful for the analyst to explore multiple hypotheses related to ACVs in the application simultaneously.

Currently, we provide filters based on following six characteristics: 1) Reachability, 2) Subsystem Interaction, 3) Presence of branch conditions that affect the resource consumption of the loop, 4) Loop Termination Patterns (LTP), 5) Monotonicity, and 6) Nesting Depth.

7.2.3 Applicability of Automated Exploration and Filters

The paper [49] reports a case study of *Gabfeed3*, a web forum software which allows users to post messages and search posted messages. The application utilizes a custom merge sort for sorting messages. The application consists of 23,882 lines of Jimple, an intermediate representation of Java bytecode.

Automated exploration creates a loop catalog consisting of all loops in the program with the following characterization for each loop: (1) the Termination Dependence Graph (TDG) and Loop Projected Control Graph (LPCG) abstractions for the loop, (2) whether the loop is monotonic, (3) applicable termination patterns, (4) subsystem APIs and the control flow paths on which they are invoked in the loop, and (5) structural characteristics such as the number of nesting levels and the inter-procedural nesting depth of the loop when the nesting is split across multiple functions.

The loop characterization can be used to create a variety of filters to select loops. This case study illustrates five filters. Using the filters, the analyst is able to isolate for further scrutiny one loop out of 112 loops in the app. On further scrutiny, an ACV was detected in this loop. The view from our filtering tool in Figure 10 shows a succession of filters that narrows down the search for a vulnerable loop.

7.3 Software Analysis

The purpose of software analysis is to produce evidence that is needed to prove or disprove the malware hypothesis developed by the analyst. For example, the hypothesis could be: GPS location is being leaked by sending it to a web client. The analysis could produce data and control flow paths from the program point where the GPS information is obtained to the point where it is sent to the web client. Another outcome could be that the information produced by the analyzer is used to verify that the GPS location is not leaked on any feasible control flow path.

Software analysis tools that work on source code use the programming language syntax and semantics to represent and reason about the possible behaviors at runtime. Tools that work on binary such as Java byte

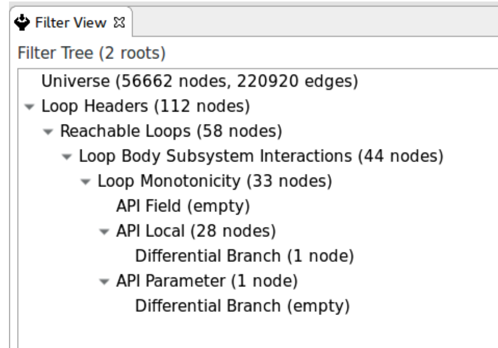


Figure 10: Using filters to select a loop likely to have an ACV

code or LLVM are useful when the source code is unavailable (e.g., analyzing proprietary COTS components). Binary analysis is often necessary to uncover potential malicious behavior introduced by an erroneous or unanticipated transformations by the compiler. Both the source code analysis [71, 27, 28, 34] and the binary analysis [7, 9, 18, 6, 57, 71, 20] are evolving research topics, with several tools being actively developed, maintained and used in practice. We discuss research directions applicable to both source code and binary analysis.

Before we discuss research directions, we make a quick note about *static* and *dynamic* analyses. The term static analysis refers to any process for analyzing code without executing it. Static analysis is powerful because it allows for the quick consideration of many possibilities. A static analysis tool can explore a large number of "what if" scenarios without having to go through all the computations necessary to execute the code for all the scenarios. Static analysis is particularly well suited to security because many security problems occur in corner cases and hard-to-reach states that can be difficult to exercise by actually running the code. The term dynamic analysis refers to any process for analyzing code by executing it. Dynamic analysis has the advantage that it can reveal the ground truth by running the code. Whereas the approximation performed by static analysis may be inaccurate and often produce false positives, static analysis can be used to narrow down the possibilities for vulnerable code. Dynamic analysis can be used to try out those possibilities dynamically to ensure that the code is indeed vulnerable. Combining static and dynamic analyses is a topic of ongoing research including *statically-informed dynamic analysis* [86] and *dynamically-informed static analysis.*

We will discuss two static analysis research directions that we find particularly important in the context of detecting CPM:

Computing Relevant Program Behaviors: All relevant program behaviors must be analyzed to verify a safety or security property. An efficient algorithm must compute the relevant behaviors directly without computing all the behaviors. This is crucial in practice because it is computationally intractable if one were to compute all behaviors to find the subset of relevant behaviors.

Interactive Graphical Static Analysis: Analyzing software for CPM inherently involves performing experiments. Static analysis is supposed to be useful to explore a large number of "what if" scenarios. However, performing such experiments is not easy in practice. There are many issues, for example, existing tools do not support on-the-fly composition of analyses to cope with open-ended "what if" possibilities. Graphs are important as a common underlying abstraction for composability of analyses.

7.4 DARPA Research: Software Analysis

Our work on DARPA APAC and STAC projects is summarized here to exemplify the software analysis research directions. The papers [86, 113, 112, 85, 72, 112, 113, 84, 49] describe our software analysis research and the relevant toolboxes we have developed as part of these projects.

7.4.1 Computing Relevant Program Behaviors

The papers [118, 119] present a mathematical foundation to define relevant behaviors. Computing the relevant program behaviors involves: (a) computing the relevant program statements, (b) computing the relevant conditions to determine the feasibility of relevant behaviors, and (c) computing the relevant program behaviors. The papers introduce the Projected Control Graph (PCG) as an abstraction to directly compute the relevant behaviors for a fairly broad class of software safety and security problems. The paper presents an efficient algorithm to transform CFG to PCG with complexity $O(|V| + |E|)$, where $|V|$ and $|E|$ are respectively numbers of nodes and edges in the CFG.

As illustrated by the following example, computing relevant behaviors is important for addressing the two fundamental software analysis challenges discussed in Section 6. In practice, the number of behaviors relevant to a task is often significantly smaller than the totality of behaviors. Computing just the relevant behaviors enables us to surmount the intractability of computing the totality of behaviors. The following example also illustrates that using PCG to compute relevant behaviors minimizes the number of path governing conditions to be analyzed for checking path feasibility.

Illustration of Relevant Program Behaviors and PCG: Consider the problem of verifying the function `foo` (Figure 11(a)) for division-by-zero (DBZ) vulnerability on line 24 which involves division by d. The CFG for `foo` is shown in Figure 11(b). The CFG is an acyclic graph with six paths. Each path yields a unique base behavior. The base behaviors are as listed in Table 1. The behaviors (B_1 and B_2) exhibit the DBZ vulnerability. The yellow highlighted statements shown in Figure 11(a) are the *relevant program statements*. In this example, these statements are relevant to the DBZ vulnerability because they affect the value of the denominator d in line 24.

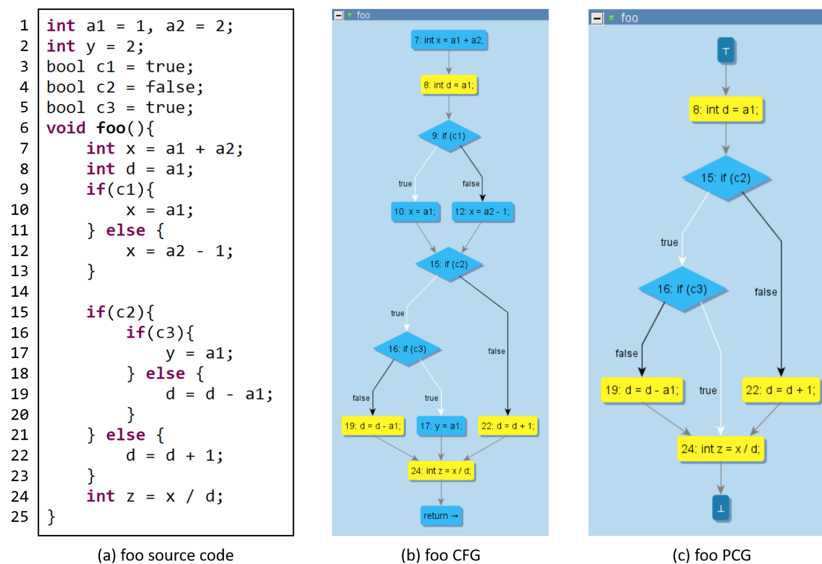


Figure 11: A division-by-zero (DBZ) vulnerability

Weiser's highly cited paper [121] on *program slice* in essence introduced the notion relevant statements. This research is crucial generalization in two important directions: (a) generalizing the notion of relevant statements for software safety and security problems, (b) generalizing from relevant statements (nodes in the CFG) to relevant behaviors (paths in the CFG).

Multiple base behaviors can be grouped so that each group corresponds to a unique *relevant behavior*. The relevant statements for the DBZ vulnerability on line 24 (Figure 11(a)) are: $8, 19, 22$ and 24. The relevant behaviors and the corresponding groups of base behaviors are listed in Table 1. Of the three relevant behaviors, RB_1 exhibits the DBZ vulnerability. Conditions $C2$ and $C3$ are included in relevant behaviors as *relevant conditions*. These conditions determine the feasibility of the vulnerable relevant behavior is RB_1.

Table 1: Base behaviors and relevant behaviors for `foo`

Base Behaviors	Relevant Behaviors
$B_1: 7, 8, 9[c_1], 10, 15[c_2], 16[\bar{c_3}], 19, 24$ $B_2: 7, 8, 9[\bar{c_1}], 12, 15[c_2], 16[\bar{c_3}], 19, 24$	$RB_1: 8, 15[c_2], 16[\bar{c_3}], 19, 24$
$B_3: 7, 8, 9[c_1], 10, 15[\bar{c_2}], 22, 24$ $B_4: 7, 8, 9[\bar{c_1}], 12, 15[\bar{c_2}], 22, 24$	$RB_2: 8, 15[\bar{c_2}], 22, 24$
$B_5: 7, 8, 9[c_1], 10, 15[c_2], 16[c_3], 17, 24$ $B_6: 7, 8, 9[\bar{c_1}], 12, 15[c_2], 16[c_3], 17, 24$	$RB_3: 8, 15[c_2], 16[c_3], 24$

The PCG for `foo` is shown in Figure 11(c). The PCG is an acyclic graph with three paths. Each path yields a unique relevant behavior. This illustration brings out that the PCG produces exactly the relevant behaviors and retains only the relevant conditions to check feasibility of the path with vulnerable behavior.

7.4.2 Interactive Graphical Static Analysis

The Eclipse and Atlas infrastructure incorporates multi-language support, a graph database, a query language, the eXtensible Common Software Graph Schema (XCSG), a variety of program analyzers, and interactive program graph visualization. This infrastructure can be used to create domain-specific toolboxes such as the Android Security Toolbox [85]. These toolboxes are written in Java and deployed as Eclipse plug-ins. The key components of Atlas infrastructure are described below.

Graph Database: After parsing a program, its complete semantics is saved as a graph with many different types of nodes and edges. Results from a number of commonly needed control and data flow analysis are incorporated in the graph database. We use attributed graphs [71] for representing program semantics. Attributes are tags for nodes and edges to represent program semantics. Program analyzers can use the *tagging mechanism* to add new semantics (e.g. call sites in C programs are tagged by an analyzer to denote use of function pointers). The tagging has multiple uses including its use for analyses to communicate with each other. As an example, we use the loop-detection analyzer to compute and tag the loop-back edges. These loop-back edges are then used by another analyzer to create an acyclic graph to compute the paths corresponding to relevant behaviors.

eXtensible Common Software Graph (XCSG) Schema: Just as high-level program languages are designed with affordances for humans, effort must be invested in designing humane representations for program analyses. Interactive analysis for human-machine collaboration must be enabled to tackle difficult problems now, and to help inspire creative solutions to automate or semi-automate verification in the future. XCSG is designed to enable interactive analysis to tackle difficult analysis problems in multi-million line programs. XCSG is designed to represent the semantics of the software, blend various common analyses, provide a basis for composing new analyses, and provide affordances required for human-machine collaboration. XCSG provides a unified approach to support for multiple programming languages. XCSG also serves as the foundation for a graphical query language. The comprehensive XCSG semantics links program text with its graphical representation as shown in Figure 12.

Graphical Query Language: The query language is implemented as an embedded domain-specific language (DSL) in Java. The rationale behind making a DSL is that it adds a layer of abstraction for expressing what to select from the graph, provides some conciseness of expression, and leaves a layer of indirection permitting query optimization. The rationale behind making it embedded is that it avoids recreating the useful features already present in an imperative language such as Java.

The query language is usually informally referred to simply as *Q*, which is the simple name of the Java interface. Q is used to describe what to select from a graph, the expression yielding a subgraph. By the builder pattern, almost all methods in the interface Q return an expression of type Q, and chaining method calls effectively specifies the query AST. A chain of Q expressions can be evaluated by calling the `Q.eval()` method, which transitions to a graph API suitable for imperative implementations. Methods of Q

```
public class T extends java.lang.Object
{
    int f;
    public void m(int)
    {
        T r0;
        int i0;
        r0 := @this: T;
        i0 := @parameter0: int;
        r0.<T: int f> = i0;
        return;
    }
}
```

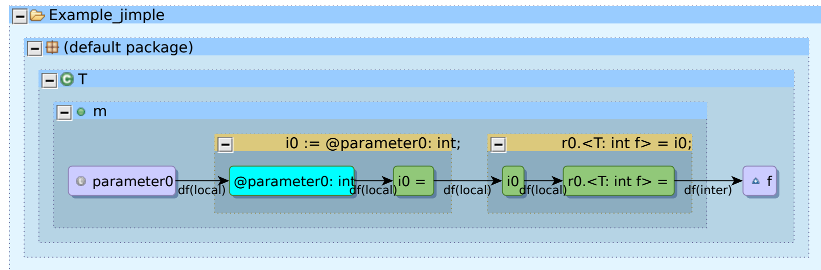


Figure 12: Linking Program Text and Graph with XCSG

are responsible for ensuring that the subgraph returned is a proper graph, where edges are present iff the incident nodes are as well. From the perspective of Q, the entire graph database is an expression called `Q.universe()`. Query results are therefore confined to returning subgraphs of the *universe*.

The primary use case behind the design of Q is enabling an analysis designer to quickly draft single line queries to select relevant portions of the graph – in essence, enabling them to "look around the corner" from their current position, and bring together related but lexically non-local elements of the program. Unlike many other graph query languages [96, 123], Q deliberately unions all matches at each step. For example, given several `tagMethod` nodes as an origin, a query for a call graph returns a single unified graph, as opposed to returning individual matches to a pattern.

In addition to interactive queries, we provide the capability to write Java programs with query APIs to write novel analyzers and verification tools. Documentation for the query APIs and XCSG are available at [3, 4, 16].

Graph Interactions with Source Correspondence: This is important for multiple resaons. For example, this capability can be used for composition of analysis. The textual queries and graph interactions can be combined. For example, for a textual query to create CFG can have *selected* as the parameter. Clicking on a node of a displayed call graph, serves as *selected* and the CFG for the selected node is displayed. Or as another example, relevant program statements can be selected by clicking on them (e.g. highlighted statements in Figure 11(a)) can produce the PCG shown in Figure 11(c). A spectrum of such capabilities can be seen in the demo videos linked with our papers [71, 85, 84].

Our research has led to the Atlas platform [71, 15] as a pluggable architecture using the Eclipse IDE. We have built various toolboxes such as the Android Security Toolbox [85] as Atlas plug-ins.

7.5 Verification

The safety and security problems are similar for the purpose of verification. A safety verification would be: verify that an event $e_1(O)$ is followed by an event $e_2(O)$ on every feasible execution path, where the two events are operations on the same object O (e.g., lock must be followed by unlock). A security verification would be: verify that an event $e_1(O)$ is *not* followed by an event $e_2(O)$ on every feasible execution path, where the two events are operations on the same object O. We shall refer to these as 2-event *matching* or *anti-matching*. Anti-matching covers software security verification defined according to the CIA triad. A confidentiality verification problem could be defined as: a *sensitive source* must *not* be followed by a *malicious sink* on any feasible execution path. Similarly, an integrity verification problem could be defined as: an *access to sensitive data* must *not* be followed by a *malicious modification to sensitive data* on any feasible execution path.

Formal verification techniques map the software verification problem to another problem that is amenable to a generic proof technique. This approach ties back to the computation complexity theory in which problems are mapped to the *satisfiability problem*. A common formal verification approach is to map the software verification problem to the *satisfiability problem* and use a SAT solver [80]. This formal approach breaks the verification into tiny steps well suited for machine execution. However, it has two major problems:

(1) it makes the verification incomprehensible to humans, and (2) it hits scalability hurdles with explosion of steps.

De Millo, Lipton, and Perlis (the first recipient of the Turing Award) [73] point to the fallacy of "absolute correctness" and argue that software verification, like "proofs" in mathematics, should provide *evidence* that humans can follow and thus be able to build trust into the correctness of the software verification. Our paper [94] on Linux verification gives examples of incorrect results by a top-notch formal verification tool.

Given the possibility of tremendous harm from CPM, we find the following research directions particularly important for software verification:

Automated Verification with Human-comprehensible Evidence: Automated verification should generate evidence alongside. Without automatically generated evidence, it can be extremely difficult and labor-intensive to cross-check the correctness of automated verification. A verification tool should convey at high-level the knowledge of specific hardness it encounters in verifying a vulnerability instance, and how that hardness is being correctly addressed by the software verification technique.

Big-step Automated Verification: Proofs in mathematics use high-level abstractions to create "big-step" proofs. The powerful machinery created by abstract algebra with its theory of groups, rings, and fields has been the foundation on which mathematician have built numerous proofs for difficult problems in number theory such as the Fermat conjecture [17]. It is inconceivable to build such proofs by simply invoking generic proof techniques devoid of deep concepts. There is a pressing need for developing "big-step" proofs for software assurance. Even for NP-complete problems, practical algorithms are designed by employing problem-specific abstractions. Big-step proofs hold the promise to render software verification proofs human-comprehensible, as well as make them efficient and scalable.

7.6 Linux Verification Research

Our verification research has evolved around 2-event matching and anti-matching that covers both software safety and security problems. We present here the work reported in papers [92, 94] to exemplify the two verification research directions.

The following verification study is done with the Berkeley Lazy Abstraction Software Verification Tool (BLAST) [52], the formal verification tool used by the Linux organization. We used BLAST to verify three recent versions (3.17-rc1, 3.18-rc1 and 3.19-rc1) of the Linux kernel. We enabled all possible `x86` build configurations via `allmodconfig` flag. The three Linux versions altogether amount to 37 million lines of code and 66,609 verification instances. The BLAST results are reported in Table 2. BLAST verifies 43,766 (65.7)% of `Lock` instances as safe, and it is inconclusive (crashes or times out) on 22,843 instances. BLAST does not find any unsafe instances.

The results reported in Table 2 are shown as: $\mathcal{C}1$ *Category* of instances verified as *safe*, $\mathcal{C}2$ *Category* of instances verified as *unsafe*, and $\mathcal{C}3$ *Category* of the remaining instances where the verification is inconclusive. Column `Type` identifies the synchronization mechanism. Columns `Locks` and `Unlocks` show the number of lock/unlock instances of each type. Note that a lock may be paired with multiple unlocks on different execution paths.

BLAST does not report any unsafe instances ($\mathcal{C}2$ *Category*) but it is inconclusive on 22,843 (34.3)% instances. BLAST does not produce evidence which makes it challenging to trace its verification to understand its failures [128].

Our study shows a practical need for evidence that shines light on why the formal verification is inconclusive. It is especially important to know if some of the 22,843 inconclusive instances are actually unsafe. The possibility of unsafe instances is worrisome for mission-critical cyber-physical systems.

BLAST pronounces 43,766 (65.7)% `Lock` instances to be safe ($\mathcal{C}1$ *Category*). Are all these instances really safe? BLAST [53] uses the *Counter Example Guided Abstraction Refinement* (CEGAR) method for verification. CEGAR does not produce a proof or other evidence to support its assertion that an instance is *safe*. While the model checker is supposed to be formally correct, it can still have false negatives (e.g., due to incorrect transformation of software into the low-level representation that the model checker requires). Without human comprehensible evidence to cross-check correctness of a formal proof, we are left with blind trust. The practical need is for evidence that makes it possible to cross-check the formal proof without

Table 2: BLAST Linux Verification Results.

Kernel	LOC	Type	Locks	Unlocks	BLAST			
					$\mathcal{C}1$	$\mathcal{C}2$	$\mathcal{C}3$	Time
3.17-rc1	12.3 M	spin	14,180	16,817	8,962 (63.2%)	0	5,218	26h
		mutex	7,887	9,497	5,494 (69.7%)	0	2,393	27h
3.18-rc1	12.3 M	spin	14,265	16,917	9,152 (64.2%)	0	5,113	30h
		mutex	7,893	9,550	5,427 (68.8%)	0	2,466	30h
3.19-rc1	12.4 M	spin	14,393	17,026	9,204 (63.9%)	0	5,189	32h
		mutex	7,991	9,653	5,527 (69.2%)	0	2,464	29h
All Kernels			**66,609**	**79,460**	**43,766 (65.7%)**	**0**	**22,843**	**173h**

having to construct a new proof starting from scratch. Without such evidence, it is practically impossible to answer the following question: Among the 43,766 instances verified as safe by BLAST, are there any cases of erroneous verification where an unsafe instance is verified as safe?

8 Conclusions

Most of the national critical infrastructure relies on industries which employ networked industrial control systems such as SCADA. Sabotage of these industries can have wide-ranging negative effects including loss of life, economic damage, property destruction, or environmental pollution.

A cyber attack is not all that different from a military attack. A cyber attacker will dedicate a significant amount of time observing and probing the target organization to find weaknesses in its defense. Any weakness found may lead to infiltration and eventually an assault.

When people think of cybersecurity today, they worry about hackers and criminals who prowl the Internet, steal people's identities, steal sensitive business information, or even steal national security secrets. Those threats are real and they exist today. But the even greater danger – the greater danger facing us in cyberspace – goes beyond crime and it goes beyond harassment. A cyber attack perpetrated by nation states or violent extremists groups could be as destructive as the 9/11 terrorist attack. Especially of concern are the communication infrastructure, the industrial control systems, and the vulnerable mission-critical software.

Most industrial control systems began as proprietary, stand-alone collections of hardware and software that were walled off from the rest of the world and isolated from most external threats. Today, widely available software applications, Internet-enabled devices and other non-proprietary IT offerings have been integrated into most such systems. This connectivity has delivered many benefits, but it also has increased the vulnerability of these systems.

A disruption in telecommunication networks can have devastating effects on all aspects of modern living, causing shortages and stoppages that ripple throughout society. Telecommunications hardware includes a vast range of products that enable communication across the entire planet, from video broadcasting satellites to telephone handsets to fiber-optic transmission cables. Services include running the switches that control the phone system, providing Internet access, and configuring private networks by which international corporations conduct business. Software makes it all work.

If a desktop operating system fails, the computer can be rebooted. If a flight control system fails, it can be a disaster with no chance to reboot the system. Malfunctioning of mission-critical software results in serious impact on business operations or upon an organization, and even can cause social turmoil and catastrophes. Mission-critical software drives online banking systems, railway and aircraft operating and control systems, electric power systems, and many other computer systems that adversely affect businesses and the society when they fail.

The real question is: *can we detect catastrophic software vulnerabilities – whether intentionally planted or not?* If we were to guard our reactors from the Stuxnet, how could we have done that? There is no escape but to create the best possible technology to analyze mission-software to discover and confirm intentional

malware or inadvertent vulnerability that could be catastrophic. But oddly enough, much of the activity that takes place under the guise of computer security is not really about solving security problems at all; it is about cleaning up the mess that security problems create. Virus scanners, firewalls, patch management, and intrusion detection systems are all means by which we make up for shortcomings in software security.

It is important that the cybersecurity research leads to practical tools and education to train a cyberforce with the necessary thinking skills to use the tools effectively. A skilled cybersecurity workforce is needed to meet the unique cybersecurity needs of critical infrastructure, enterprise, and operational technology systems and networks.

Acknowledgements

We thank our colleagues from Iowa State University and EnSoft for their help with this paper. We are grateful to Jeremias Sauceda and Nikhil Ranade for their significant contributions to our research on graphical software analysis and verification. Tom Deering, Eric Woestman, Theodore Murdock, Shrawan Kumar, Akshay Deepak, and Damanjit Singh have played important roles in evolving the research. Dr. Kothari is the founder President and a financial stakeholder in EnSoft.

References

[1] Android intent. https://stackoverflow.com/questions/6578051/what-is-an-intent-in-android. (Accessed on 01/22/2018).

[2] Android intent (android developer guide). https://developer.android.com/guide/components/intents-filters.html. (Accessed on 01/22/2018).

[3] Atlas queries documentation. http://www.ensoftcorp.com/atlas_docs/javadoc/2.x/index.html?com/ensoftcorp/atlas/core/query/Q.html. (Accessed on 01/22/2018).

[4] Atlas wiki. http://ensoftatlas.com/wiki/Main_Page. (Accessed on 01/22/2018).

[5] Automated program analysis for cybersecurity (apac). http://www.defenseinnovationmarketplace.mil/resources/DARPA%202011%208%203%20APAC%20Industry%20Day.pdf. (Accessed on 01/22/2018).

[6] Bap - a binary analysis platform. https://www.grammatech.com/products/codesonar. (Accessed on 04/02/2018).

[7] Bitblaze. http://bitblaze.cs.berkeley.edu. (Accessed on 04/02/2018).

[8] Blueborne cyber threat impacts amazon echo and google home. https://www.armis.com/blueborne-cyber-threat-impacts-amazon-echo-google-home/. (Accessed on 01/22/2018).

[9] Codesonar. https://www.grammatech.com/products/codesonar. (Accessed on 04/02/2018).

[10] Coverity static analysis, static application security testing. https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html?utm_source=google&utm_medium=paid%20search&utm_term=coverity&utm_campaign=G_S_Coverity_Exact&cmp=ps-SIG-G_S_Coverity_Exact&gclid=EAIaIQobChMIhe3pqvXr2AIVV7bACh1fEgaQEAAYASAAEgL8K_D_BwE. (Accessed on 01/22/2018).

[11] Cve - common vulnerabilities and exposures (cve). https://cve.mitre.org/. (Accessed on 01/22/2018).

[12] Cvss v3.0 specification document. https://www.first.org/cvss/specification-document. (Accessed on 01/22/2018).

[13] Ddos attack that disrupted internet was largest of its kind in history, experts say — technology — the guardian. https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet. (Accessed on 01/22/2018).

[14] Department of defense - directive number 3020.40. http://policy.defense.gov/Portals/11/Documents/hdasa/newsletters/302040p.pdf. (Accessed on 01/22/2018).

[15] Ensoft corp. http://www.ensoftcorp.com. (Accessed on 01/22/2018).

[16] Extensible common software graph. http://ensoftatlas.com/wiki/Extensible_Common_Software_Graph. (Accessed on 01/22/2018).

[17] Fermat conjecture. https://en.wikipedia.org/wiki/Fermat%27s_Last_Theorem. (Accessed on 01/22/2018).

[18] Grackle. https://grackle.galois.com. (Accessed on 04/02/2018).

[19] Hp fortify. http://www.ndm.net/sast/hp-fortify. (Accessed on 01/22/2018).

[20] Klee llvm execution engine. http://klee.github.io/. (Accessed on 04/02/2018).

[21] Modbus. http://www.modbus.org. (Accessed on 01/22/2018).

[22] Modbus penetration testing framework. https://github.com/enddo/smod. (Accessed on 01/22/2018).

[23] Morris worm - wikipedia. https://en.wikipedia.org/wiki/Morris_worm. (Accessed on 01/22/2018).

[24] National vulnerability database. https://nvd.nist.gov/. (Accessed on 01/22/2018).

[25] Open ssl developer confesses to causing heartbleed bug — daily mail online. http://www.dailymail.co.uk/sciencetech/article-2602277/Heartbleed-accident-Developer-confesses-coding-error-admits-effect-clearly-severe.html#ixzz546wC2cbw. (Accessed on 01/22/2018).

[26] Sandvines packetlogic devoices used to deploy government spyware in turkey and redirect egyptian users to affiiliate ads. https://citizenlab.ca/2018/03/bad-traffic-sandvines-packetlogic-devices-deploy-government-spyware-turkey-syria/. (Accessed on 04/02/2018).

[27] Slam. https://www.microsoft.com/en-us/research/project/slam/. (Accessed on 04/02/2018).

[28] Soot. https://github.com/Sable/soot. (Accessed on 04/02/2018).

[29] Space/time analysis for cybersecurity (stac). https://www.darpa.mil/program/space-time-analysis-for-cybersecurity. (Accessed on 01/22/2018).

[30] Splint (programming tool) - wikipedia. https://en.wikipedia.org/wiki/Splint_(programming_tool). (Accessed on 01/22/2018).

[31] Stuxnet - wikipedia. https://en.wikipedia.org/wiki/Stuxnet. (Accessed on 01/22/2018).

[32] Sy110: Phases of a cyber attack / cyber recon. https://www.usna.edu/CyberDept/sy110/lec/cyberRecon/lec.html. (Accessed on 01/22/2018).

[33] Telecommunications equipment - wikipedia. https://en.wikipedia.org/wiki/Telecommunications_equipment. (Accessed on 01/22/2018).

[34] Wala. http://wala.sourceforge.net/wiki/index.php/Main_Page. (Accessed on 04/02/2018).

[35] Wannacry ransomware attack. https://arstechnica.com/information-technology/2017/05/an-nsa-derived-ransomware-worm-is-shutting-down-computers-worldwide/. (Accessed on 01/22/2018).

[36] Final report on the august 14, 2003 blackout in the united states and canada: Causes and recommendations. https://energy.gov/sites/prod/files/oeprod/DocumentsandMedia/BlackoutFinal-Web.pdf, April 2004. (Accessed on 01/22/2018).

[37] 2009 cyberspace policy review — homeland security. https://www.dhs.gov/sites/default/files/publications/Cyberspace_Policy_Review_final_0.pdf, 2009. (Accessed on 01/22/2018).

[38] Defense contractors northrop grumman, l-3 communications hit by cyber-attack. https://www.cioinsight.com/c/a/Latest-News/Defense-Contractors-Northrop-Grummond-L3-Communications-Hit-by-CyberAttack-106322, June 2011. (Accessed on 01/22/2018).

[39] National cyber security divisions control systems security program (cssp). https://ics-cert.us-cert.gov/sites/default/files/recommended_practices/DHS_Common_Cybersecurity_Vulnerabilities_ICS_2010.pdf, May 2011. (Accessed on 01/22/2018).

[40] Investigative report on the u.s. national security issues posed by chinese telecommunications companies huawei and zte. https://intelligence.house.gov/sites/intelligence.house.gov/files/documents/huawei-zte%20investigative%20report%20(final).pdf, October 2012. (Accessed on 01/22/2018).

[41] Darpa-baa-13-11: Vetting commodity it software and firmware (vet), updated. https://govtribe.com/project/darpa-baa-13-11-vetting-commodity-it-software-and-firmware-vet, Februrary 2013. (Accessed on 01/22/2018).

[42] Industroyer: Biggest threat to industrial control systems since stuxnet. https://www.welivesecurity.com/2017/06/12/industroyer-biggest-threat-industrial-control-systems-since-stuxnet/, June 2017. (Accessed on 01/22/2018).

[43] National initiative for cybersecurity education (nice) cybersecurity workforce framework. https://csrc.nist.gov/csrc/media/publications/sp/800-181/archive/2016-11-02/documents/sp800_181_draft.pdf, August 2017. (Accessed on 01/22/2018).

[44] Threat modeling cheat sheet - owasp. https://www.owasp.org/index.php/Threat_Modeling_Cheat_Sheet, December 2017. (Accessed on 01/22/2018).

[45] RTCA (Firm). SC 167. *Software considerations in Airborne Systems and equipment certification*. RTCA, Incorporated, 1992.

[46] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, and tools*, volume 2. Addison-wesley Reading, 2007.

[47] Jafar M. Al-Kofahi, Suresh Kothari, and Christian Kästner. Four languages and lots of macros: Analyzing autotools build systems. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2017, pages 176–186. ACM, 2017.

[48] Keith Alexander. Keynote - 2011 cyber & space symposium. https://www.youtube.com/watch?v=jaaU5nGDh68, November 2011. (Accessed on 01/22/2018).

[49] Payas Awadhutkar, Ganesh Ram Santhanam, Benjamin Holland, and Suresh Kothari. Intelligence amplifying loop characterizations for detecting algorithmic complexity vulnerabilities. In *The 24th Asia-Pacific Software Engineering Conference (APSEC 2017)*, 2017.

[50] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *CoRR*, abs/1610.00502, 2016.

[51] Dirk Beyer. Status report on software verification. In *TACAS*, volume 8413, pages 373–388, 2014.

[52] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[53] Dirk Beyer and Alexander K. Petrenko. Linux driver verification. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, pages 1–6, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[54] Wayne Boyer and Miles McQueen. Ideal based cyber security technical metrics for control systems. In *International Workshop on Critical Information Infrastructures Security*, pages 246–260. Springer, 2007.

[55] Frederick P. Brooks, Jr. The computer scientist as toolsmith ii. *Commun. ACM*, 39(3):61–68, March 1996.

[56] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. *Botnet Detection*, pages 65–88, 2008.

[57] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[58] Eric Byres and Justin Lowe. The myths and facts behind cyber security risks for industrial control systems. In *Proceedings of the VDE Kongress*, volume 116, pages 213–218, 2004.

[59] C. Canal and A. Idani. *Software Engineering and Formal Methods: SEFM 2014 Collocated Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, WS-FMDS, Grenoble, France, September 1-2, 2014, Revised Selected Papers*. Lecture Notes in Computer Science. Springer International Publishing, 2015.

[60] Anton Cherepanov. Win32/industroyer a new threat for industrial control systems. `https://www.welivesecurity.com/wp-content/uploads/2017/06/Win32_Industroyer.pdf`, June 2017. (Accessed on 01/22/2018).

[61] Alonzo Church. A note on the entscheidungsproblem. *The journal of symbolic logic*, 1(1):40–41, 1936.

[62] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[63] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification*, pages 1–30. Springer, 2012.

[64] Darren Cofer. Model checking: cleared for take off. *Model Checking Software*, pages 76–87, 2010.

[65] Zachary A Collier, Mahesh Panwar, Alexander A Ganin, Alexander Kott, and Igor Linkov. Security metrics in industrial control systems. In *Cyber-security of SCADA and Other Industrial Control Systems*, pages 167–185. Springer, 2016.

[66] Douglas Comer. *Operating system design: the Xinu approach, Linksys version*. CRC Press, 2011.

[67] Lucian Constantin. Flame authors order infected computers to remove all traces of the malware - cio. `https://www.cio.com.au/article/427005/flame_authors_order_infected_computers_remove_all_traces_malware/`, June 2012. (Accessed on 01/22/2018).

[68] Scott Crosby. Denial of service through regular expressions. *Usenix Security work in progress report*, 2003.

[69] John D'Arcy and Gwen Greene. Security culture and the employment relationship as drivers of employees security compliance. *Information Management & Computer Security*, 22(5):474–489, 2014.

[70] Richard A De Millo, Richard J Lipton, and Alan J Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, 1979.

[71] Tom Deering, Suresh Kothari, Jeremias Sauceda, and Jon Mathews. Atlas: a new way to explore software, build analysis tools. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 588–591. ACM, 2014.

[72] Tom Deering, Ganesh Ram Santhanam, and Suresh Kothari. Flowminer: Automatic summarization of library data-flow for malware analysis. In *International Conference on Information Systems Security*, pages 171–191. Springer, 2015.

[73] RA DeMillo, RJ Lipton, and AJ PerHls. Social processes and proofs of programs and theorems. In *Proc. Fourth ACM Symposium on Principles of Program-ming Languages*, pages 206–214, 1979.

[74] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. Evil pickles: Dos attacks based on object-graph engineering (artifact). In *DARTS-Dagstuhl Artifacts Series*, volume 3. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[75] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *ACM SIGPLAN Notices*, volume 43, pages 270–280. ACM, 2008.

[76] Paul Ducklin. Anatomy of a goto fail apples ssl bug explained, plus an unofficial patch for os x! naked security. `https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/`, February 2014. (Accessed on 01/22/2018).

[77] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 43–56. USENIX Association, 2005.

[78] Kathleen Fisher. High assurance cyber military systems (hacms). `http://www.cyber.umd.edu/sites/default/files/documents/symposium/fisher-HACMS-MD.pdf`, May 2013. (Accessed on 01/22/2018).

[79] National Institute for Standards and Technology (NIST). Nist guide to industrial control systems security. `http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-82r2.pdf`, May 2015. (Accessed on 01/22/2018).

[80] Malay Ganai and Aarti Gupta. *SAT-based scalable formal verification solutions.* Springer, 2007.

[81] Michael R Garey and David S Johnson. Computers and intractability. a guide to the theory of np-completeness. a series of books in the mathematical sciences, 1979.

[82] Allen Goldberg, Tie-Cheng Wang, and David Zimmerman. Applications of feasible path analysis to program testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 80–94. ACM, 1994.

[83] Andy Greenberg. Hackers remotely kill a jeep on the highwaywith me in it — wired. `https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/`, July 2015. (Accessed on 01/22/2018).

[84] Benjamin Holland, Payas Awadhutkar, Suresh Kothari, Ahmed Tamrawi, and Jon Mathews. Comb: Computing relevant program behaviors. In *International Conference on Software Engineering Demonstration track*, page To appear., 2018.

[85] Benjamin Holland, Tom Deering, Suresh Kothari, Jon Mathews, and Nikhil Ranade. Security toolbox for detecting novel and sophisticated android malware. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 733–736. IEEE Press, 2015.

[86] Benjamin Holland, Ganesh Ram Santhanam, Payas Awadhutkar, and Suresh Kothari. Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*, pages 79–84. IEEE, 2016.

[87] Michael Howard and David LeBlanc. *Writing secure code.* Pearson Education, 2003.

[88] IBM. Security attacks on industrial control systems - managed security services research report. https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=SEL03046USEN. (Accessed on 01/22/2018).

[89] Kamal Jabbour and Sarah Muccio. The science of mission assurance. *Journal of Strategic Security*, 4(2):61, 2011.

[90] Michael B Kelley. Stuxnet was far more dangerous than previous thought - business insider. http://www.businessinsider.com/stuxnet-was-far-more-dangerous-than-previous-thought-2013-11, November 2013. (Accessed on 01/22/2018).

[91] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[92] S. Kothari, P. Awadhutkar, and A. Tamrawi. Insights for practicing engineers from a formal verification study of the linux kernel. In *2016 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 264–270, Oct 2016.

[93] S. Kothari, A. Deepak, A. Tamrawi, B. Holland, and S. Krishnan. A human-in-the-loop approach for resolving complex software anomalies. In *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1971–1978, Oct 2014.

[94] Suresh Kothari, Payas Awadhutkar, Ahmed Tamrawi, and Jon Mathews. Modeling lessons from verifying large software systems for safety and security. In *2017 Winter Simulation Conference (WSC)*, pages 1431–1442, 2017.

[95] Vadim Kotov and Fabio Massacci. Anatomy of exploit kits. *Engineering Secure Software and Systems*, 7781:181–196, 2013.

[96] Mahesh Lal. *Neo4j Graph Data Modeling.* Packt Publishing Ltd, 2015.

[97] Sihyung Lee. *Reducing Complexity of Large-scale Network Configuration Management.* PhD thesis, Pittsburgh, PA, USA, 2010. AAI3415822.

[98] J. L. LIONS. Ariane 5 failure - full report. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html, July 1996. (Accessed on 01/22/2018).

[99] Alan K Mackworth. Constraint satisfaction problems. *Encyclopedia of AI*, 285:293, 1992.

[100] P MELL. A complete guide to the common vulnerability scoring system version 2.0. *http://www.first.org/cvss/cvss-guide.pdf*, 2007.

[101] Arash Nourian and Stuart Madnick. A systems theoretic approach to the security threats in cyber physical systems applied to stuxnet. *IEEE Transactions on Dependable and Secure Computing*, 2015.

[102] Ebenezer A Oladimeji, Sam Supakkul, and Lawrence Chung. Security threat modeling and analysis: A goal-oriented approach. In *Proc. of the 10th IASTED International Conference on Software Engineering and Applications (SEA 2006)*, pages 13–15, 2006.

[103] Leon E. Panetta. Defense.gov transcript: Remarks by secretary panetta on cybersecurity to the business executives for national security, new york city. http://archive.defense.gov/transcripts/transcript.aspx?transcriptid=5136, October 2012. (Accessed on 01/22/2018).

[104] Fabio Pasqualetti, Florian Dörfler, and Francesco Bullo. Attack detection and identification in cyber-physical systems. *IEEE Transactions on Automatic Control*, 58(11):2715–2729, 2013.

[105] Alin C Popescu, Brian J Premore, and Todd Underwood. Anatomy of a leak: As9121. *Renesys Corp., http://www. renesys. com/tech/presentations/pdf/renesys-nanog34. pdf*, 2005.

[106] Kevin Poulsen. Slammer worm crashed ohio nuke plant network. `https://www.securityfocus.com/news/6767`, August 2003. (Accessed on 01/22/2018).

[107] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on programming*, pages 337–351. Springer, 1982.

[108] Jean-Pierre Queille and Joseph Sifakis. Fairness and related properties in transition systemsa temporal logic to deal with fairness. *Acta Informatica*, 19(3):195–220, 1983.

[109] Brian Randell. The origins of computer programming. *IEEE Annals of the History of Computing*, 16(4):6–14, 1994.

[110] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[111] DAVID E. SANGER. Obama ordered wave of cyberattacks against iran - the new york times. `http://www.nytimes.com/2012/06/01/world/middleeast/obama-ordered-wave-of-cyberattacks-against-iran.html?pagewanted=1&_r=1&hp`, June 2012. (Accessed on 01/22/2018).

[112] Ganesh Ram Santhanam, Benjamin Holland, Suresh Kothari, and Jon Mathews. Interactive visualization toolbox to detect sophisticated android malware. In *Visualization for Cyber Security (VizSec), 2017 IEEE Symposium on*, pages 1–8. IEEE, 2017.

[113] Ganesh Ram Santhanam, Benjamin Holland, Suresh Kothari, and Nikhil Ranade. Human-on-the-loop automation for detecting software side-channel vulnerabilities. In *International Conference on Information Systems Security*, pages 209–230. Springer, 2017.

[114] Bruce Schneier. Heartbleed - schneier on security. `https://www.schneier.com/blog/archives/2014/04/heartbleed.html`, April 2014. (Accessed on 01/22/2018).

[115] Tony Smith. Hacker jailed for revenge sewage attacks. `http://www.theregister.co.uk/2001/10/31/hacker_jailed_for_revenge_sewage/`, October 2001. (Accessed on 01/22/2018).

[116] Panos Stratis. Formal verification in large-scaled software: Worth to ponder. `https://blog.inf.ed.ac.uk/sapm/2014/02/20/formal-verification-in-large-scaled-software-worth-to-ponder/`, 2014. (Accessed on 01/25/2018).

[117] Frank Swiderski and Window Snyder. *Threat Modeling (Microsoft Professional)*, volume 7. Microsoft Press, 2004.

[118] Ahmed Tamrawi and Suresh Kothari. Projected control graph for accurate and efficient analysis of safety and security vulnerabilities. In *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific*, pages 113–120. IEEE, 2016.

[119] Ahmed Tamrawi and Suresh Kothari. Projected control graph for computing relevant program behaviors. *Journal of Science of Computer Programming*, To appear.

[120] Alan M. Turing. The use of dots as brackets in church's system. *The Journal of Symbolic Logic*, 7(4):146–156, 1942.

[121] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[122] D. E. Whitehead, K. Owens, D. Gammel, and J. Smith. Ukraine cyber-induced power outage: Analysis and practical mitigation strategies. In *2017 70th Annual Conference for Protective Relay Engineers (CPRE)*, pages 1–8, April 2017.

[123] Peter T Wood. Query languages for graph databases. *ACM SIGMOD Record*, 41(1):50–60, 2012.

[124] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys (CSUR)*, 41(4):19, 2009.

[125] Avishai Wool. A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67, 2004.

[126] Victoria Woollaston. Open ssl developer confesses to causing heartbleed bug — daily mail online. `http://www.dailymail.co.uk/sciencetech/article-2602277/Heartbleed-accident-Developer-confesses-coding-error-admits-effect-clearly-severe.html#ixzz546xyGkwC`, April 2014. (Accessed on 01/22/2018).

[127] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3):16, 2007.

[128] Ilja S Zakharov, Mikhail U Mandrykin, Vadim S Mutilin, EM Novikov, Alexander K Petrenko, and Alexey V Khoroshilov. Configurable toolset for static verification of operating systems kernel modules. *Programming and Computer Software*, 41(1):49–64, 2015.

[129] Jian Zhang and Xiaoxu Wang. A constraint solver and its application to path feasibility analysis. *International Journal of Software Engineering and Knowledge Engineering*, 11(02):139–156, 2001.