TRUST IN SOFT

WHITE PAPER

From Bare Metal to Kernel Code:

How Exhaustive Static Analysis Can Guarantee Airtight Security and Reliability in Low-level Software and Firmware

December 2022

Table of contents

LOW LEVEL CODE

Introduction

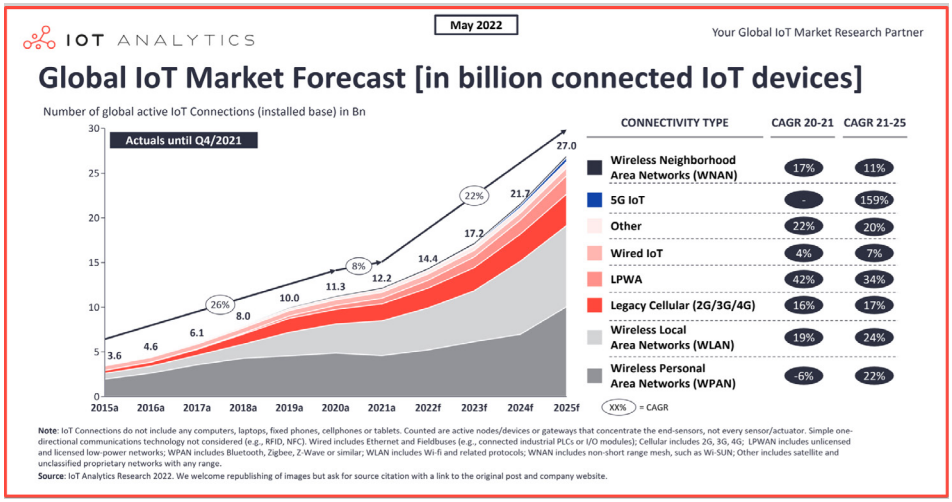


Figure 1: Source: State of IoT 2022: Number of connected IoT devices

The proliferation of Internet-connected devices has exploded over the past decade. Today, we find them everywhere.

Our smartphones, tablets, and PCs… TVs, stereos, gaming consoles, and other home entertainment systems… our printers and many other office devices all connect to the Internet through Wi-Fi or cellular networks. So do many of our home appliances as well as the growing number of smart-home and smart-building technologies. We find similar devices in practically every room of every establishment we visit. We even find them along the street.

Embedded processors and wireless transmitters and receivers are installed in everything from public Wi-Fi hotspots to self-service fuel pumps. They're in safety-critical and security-critical IoT devices embedded in automobiles, medical devices, and point-of-sale terminals. They're in systems for factory automation, energy distribution, lighting, safety, security, and surveillance. They're flying above us in all types of aircraft, from passenger liners to drones to satellites.

According to IoT business intelligence provider IoT Analytics, worldwide IoT connections were up 8% in 2021 and were expected to grow by 18% in 2022. The firm has forecast that this expansion should continue for the foreseeable future, as illustrated in Figure 1. Such growth will continue to create new opportunities and benefits for technology providers and consumers.

Unfortunately, this ever-expanding web of interconnected devices is also creating a serious security challenge for the technology industry.

The low-level code in these devices—the code that interfaces directly with the hardware, like operating system kernels, device firmware, and drivers and controllers—is, along with the hardware itself, the foundation of cybersecurity.

These low-level layers in the firmware/software stack have access to both the hardware below and the application layers above. If they aren't secure, then neither is anything that sits on top of them.

Coding flaws in low-level code create vulnerabilities that hackers exploit. Once hackers gain access, they can either take control of the device or access the data stored within. For owners of these devices that could mean the theft of their sensitive data—possibly their bank or credit card information.

Bugs in the low-level software such as buffer overflows, or non-initialized variables also have a significant reliability impact. They can cause the software to crash or introduce non-deterministic behavior, which can impact the supplier's image. In the case of safety-critical systems like automobiles, and medical devices, the result could be the injury or death of passengers or patients, product recalls, lawsuits, and other losses. Whatever the target markets, even in markets that are not seen as reliability critical, it's not always possible nor convenient to remotely update software in the field.

Verifying that low-level code is free from coding errors and vulnerabilities is a serious challenge. Standard software verification methods like traditional static analysis and software testing are not up to the task of fully securing today's connected devices; they cannot provide a guarantee that all vulnerabilities have been eliminated.

Fortunately, there is an available alternative that can provide such a guarantee.

Exhaustive static analysis enables developers to find and eliminate 100% of undefined behaviors (defects like buffer overflow, uninitialized memory access, etc.) that can leave low-level code vulnerable to attack or result in software crashing or non-deterministic behavior. It gives device manufacturers and their customers an iron-clad guarantee that their products are completely free of such defects and vulnerabilities.

In the remainder of this white paper, we will examine in greater detail:

- The challenges of ensuring the security, and reliability of low-level code in today's environment,
- Why traditional code verification methods are not up to these challenges, and
- How exhaustive static analysis is able to meet those same challenges and guarantee cybersecurity and reliability in low-level code.

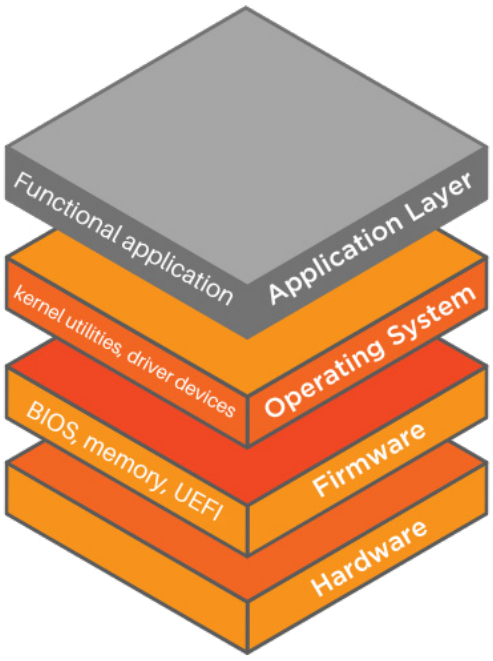
We'll start by answering a simple question.

What is "low-level" code?

Low-level code can be found in every connected device. It's not the code in user applications like Microsoft Office, Adobe Acrobat, or the apps you download from Apple's App Store and Google Play for use on your phone. Instead, it's the code used in the layers below those applications—the layers that interact directly with the device hardware.

In PCs, tablets, smartphones, and other devices that run an operating system (OS) like Windows, iOS, Android, or Linux, low-level code is used in the OS kernel and in firmware applications like secure boot, firmware update, device recovery attestation, and the system BIOS.

Low-level code is also used in so-called "bare metal" applications—programs that run directly on the hardware without the assistance of an OS. Bare metal software is common in small-scale, low-power, and memory-limited devices like medical implants, remote IoT devices, and spacecraft instrumentation. It's used frequently in safety-critical applications in the aerospace, defense, and automotive industries, as well as in time-critical applications like RTOS kernels.



LOW LEVEL CODE

An inviting target for hackers

Because low-level code offers access to both the hardware below and the high-level code above, any flaws in it make inviting targets for hackers. Exploits against those flaws could cause the hardware to crash or allow the attacker to gain control of the high-level code.

A global security survey conducted by Microsoft found that 80% of enterprises have experienced at least one firmware attack in the past two years[2].Meanwhile, the National Institute of Standards and Technology's (NIST) continually updated National Vulnerability Database (NVD) has shown a better than five-fold increase in firmware attacks since 2017[3].

"These attacks are of particular importance," says firmware and hardware protection firm Eclypsium, "because they enable attackers to gain fundamental control of enterprise devices, subvert security controls, and persist invisibly, undetected by traditional security solutions."[4]

Operating systems are even more cybersecurity-critical. They have far greater reach than firmware. An OS has access to every application that runs on top of it. If the OS is compromised, all those applications could be compromised as well. For this reason, Microsoft has called the OS kernel "an emerging gap in (cybersecurity) defense."

The following example illustrates just how dangerous that gap can be.

In 2017, Wikileaks revealed that the CIA had "weaponized" numerous zero-day vulnerabilities in iPhones, Google Android, Microsoft Windows, Samsung smart TVs, vehicle control systems, and other devices. These were vulnerabilities they had either discovered, developed, obtained from other agencies, or purchased from cyber arms contractors.

A zero-day (or 0-day) is a software vulnerability previously unknown to those who should be interested in its mitigation, like the software vendor. Until the vulnerability is mitigated, hackers can exploit it to adversely affect programs, data, additional computers, or a network.

Wikileaks also revealed that the CIA had subsequently lost control of its zero-day exploit arsenal through unauthorized circulation among former government hackers and contractors. That arsenal had thus become available to hostile governments, cyber mafia, and malevolent hackers worldwide. Until all those target zero-days are mitigated, millions of devices are at risk.

In order to reduce such risks, some OS designers adopt modular architectures, using hypervisors for example. Because you have several operating systems running in parallel but isolated from one another under the supervision of the hypervisor, modularity can limit the spread of a malware infection and keep your system running.

> ***That type of risk reduction, however, can only go so far. If there are exploitable flaws in your hypervisor, such security measures could all be for naught. Far better to have no zero-day vulnerabilities whatsoever in your code.***

LOW LEVEL CODE

Embedded systems (and businesses) at high risk

According to the consulting firm RSK Cyber Security, embedded systems are particularly prone to cyberattacks[5].

For businesses, this can present a serious risk, as these devices are directly interconnected with the core network of the company. A coding error in an embedded device can provide an avenue for an attack on the enterprise as a whole. The flaw not only compromises the device; it could take down the company's entire network.

There are several reasons embedded systems are so susceptible. First, they can be attacked through vulnerabilities on two fronts, through both the hardware and the code (software and firmware). Second, integration with the IoT (connectivity) increases the number of attack vectors.

"Another reason is stuffing a small embedded system with many functionalities leads to a lack of security by design," says Praveen Joshi of RSK Cyber Security[6].

To save memory space and limit power consumption, developers of these embedded applications often resort to non-standard coding structures While done out of necessity, this practice results in optimized code that makes bugs hard to find.

COMMON VULNERABILITIES AND RELIABILITY DEFECTS IN LOW-LEVEL CODE

Many attacks on embedded systems target vulnerabilities caused by bugs in low-level code.

One of the most common types of attacks against embedded firmware and software targets a coding error vulnerability known as memory buffer overflow. This software weakness was ranked #1 on the CWE Top 25 2019 list[7].It typically ranks highly from year to year and is most prevalent in the C and C++ programming languages.

"In this type of attack, hackers exploit the system vulnerabilities to swamp the device's memory," says Joshi. "Attackers manually fill the memory buffer allocated to contain the moving data inside the embedded systems. The OS of the embedded system will attempt to record some data in the memory section next to the buffer. But, eventually, it will fail[8]."

Other dangerous undefined-behavior vulnerabilities include:

•Integer overflow errors

•Integer underflow errors

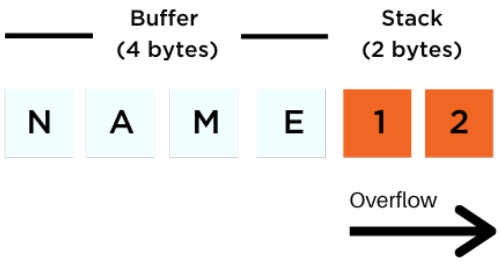
•Buffer overwrite errors

•Buffer overread errors

•Null pointer deference errors

These common coding errors also have significant impact on overall reliability and quality of the product. Software reliability is an important factor in software quality alongside security, performance and availability. Software reliability is hard to achieve due to high levels of complexity. In order to achieve this, an acceptable level of reliability should be specified and the software should be tested. This implies the generation of a set of test data corresponding to the desired reliability level. It can be very difficult to this exhaustively at a reasonable cost.

How can software development organizations protect their products against such exploits and reliability defects?



LOW LEVEL CODE

Traditional analysis and testing are not the answer

The two standard solutions for software verification and bug removal—and still the most common methods used today by the majority of software and systems developers—are traditional static code analysis and software testing.

Unfortunately, both these methods have shortcomings that are magnified when applied to embedded systems and other low-level code applications.

DRAWBACKS OF TRADITIONAL STATIC ANALYSIS

Unlike applications that run atop an operating system, low-level code doesn't have the support of an abstracted, generic platform created by an operating system. It must take into account the specifics of the hardware on which it runs and any restrictions that hardware presents, such as power consumption constraints or memory limitations. Code for embedded systems often has to meet very stringent timing requirements as well.

For those reasons, low-level code often can't conform to coding standards built for upper-layer applications. What's more, low-level code accesses memory in a manner that is quite different from that of higher-level (abstracted) applications. Traditional code analysis tools are generally not equipped to deal with either of these factors. As a result, they frequently yield a high volume of false positives and false negatives when applied to such code.

False positives

Traditional static analysis is based on a set of rules that the static analysis tool expects code to follow. These rules include standards of what is considered good coding structure. In a static analysis context, a "false positive" occurs when the static analysis tool incorrectly reports that one of its rules was violated.

Since low-level programmers must account for the particulars of their target hardware and tend to stray frequently from the rules of good coding, low-level code is prone to high volumes of false positives when conventional static analysis tools and techniques are applied to it.

False positives tend to annoy developers because they slow progress, increase the tedium of the job, and waste precious time. They force programmers to investigate issues that turn out to be unimportant.

Developers get bored very quickly with verifying errors flagged by their static analysis tools. The tedium of spending days investigating large numbers of false alarms can often lead them to dismiss some warnings as false positives when they are, in fact, true bugs. They thus compromise the integrity and security they've been trying to build into their system.

False negatives

"False negatives" are undefined behaviors (bugs) that are missed and therefore not flagged by the analysis tool.

Since the structure of low-level code is often complicated due to its hardware constraints, it may contain errors that traditional static analysis tools are not programmed to recognize. Some of these bugs may require a significant amount of calculation to reveal—calculations that are omitted from traditional static analysis tools in the interest of returning results very quickly.

Thus, once you've managed to correct all the bugs and verify all the false positives your static analysis tool has found, you may be left with a false sense of security. In reality, this is a very dangerous feeling. Your tool has given you the green light, but there may still be dozens or even hundreds of bugs in your code. Some of them could be very serious.

In a critical embedded system, these unflagged errors—these false negatives—could be disastrous for both the system manufacturer and their customer, as they were in cases like:

- ***The WhatsApp Integer Overflow[9],***
- ***Toyota's unintended acceleration firmware problem[10],***
- ***Smiths Medical's Medfusion 4000 Wireless Syringe Infusion Pump [11], and***
- ***The Boeing 787 integer overflow error[12].***

Drawbacks of traditional software testing

Like traditional static analysis, software testing also suffers from two major drawbacks when used to verify low-level code, especially code that must be either highly reliable or highly secure.

The first of these drawbacks is the length of the testing process.

Traditional software testing relies on defining test cases that account for as many operational scenarios as possible. You then run tests until you either (1) cover all your scenarios, or (2) run out of time. The latter tends to be the more frequent case.

For complex code, however, the number of possible test cases—i.e. the number of possible input and state combinations—can be astronomical. Even a vaguely representative subset of those cases could require more time than the project schedule and budget will allow.

The second drawback, highly related to the first, is test case coverage.

You may have an automated test campaign that tests for millions of input value combinations, but still, you can never test every combination because there are simply too many. Even when you stop finding errors, you're never sure if you've tested enough.

So, just as you don't know how many bugs your static analysis tool failed to flag, you don't know how many of those bugs also slipped past your testing campaign.

Each of the drawbacks just discussed presents a risk many organizations cannot afford to take. They would be exposing their customers to potential dangers which are difficult to predict. As a consequence, they would be exposing their own company to costly product recalls, prolonged loss of revenue, potential lawsuits, and long-term brand reputation damage.

> ***So, again, how can companies protect themselves? What can they use instead?***

LOW LEVEL CODE

Exhaustive static analysis and its zero-defect guarantee

For many, the answer is exhaustive static analysis.

Exhaustive static analysis is an alternative to traditional static analysis. Rather than sets of rules, exhaustive static analysis uses mathematical formal methods to prove unequivocally that your code is free from coding errors and undefined behaviors that hackers can exploit, or which could impact overall product reliability.

The method was initially developed for formal verification of safety-critical systems, like those of the aerospace industry, where a software failure could result in the destruction of property and the loss of human life. Having evolved over nearly two decades of refinement, it is now approved for use in place of software testing for the certification of airborne systems under DO-178C, the aerospace industry's de facto standard for software certification[13].

Exhaustive static analysis is a methodology that makes use of a variety of formal methods to answer the questions users ask about their code. It takes know-how developed to guarantee the behavior of safety-critical systems and expands its scope to guarantee data cybersecurity as well as safety and functionality. It makes that know-how available to all developers for use in the development of any device—from smartphones to game consoles, from medical technology to remote monitoring devices.

Exhaustive static analysis is also a framework where a broad range of formal methods collaborate as one. Its tools contain algorithms that choose the right formal method according to the question being asked. A good exhaustive static analysis tool can switch seamlessly from one formal method to another, depending on the problem it has been asked to solve.

In other words, exhaustive static analysis is a holistic approach, not one of brute force. Rather than trying to solve every problem using a single formal method, the framework has been designed to determine which formal method or combination of formal methods is best suited to solving the problem at hand and to apply those methods in the best way possible.

LOW LEVEL CODE

Ideally suited to low-level code

Software for safety-critical aerospace apps historically ran on bare metal—directly on the hardware—with no OS in between. This was done primarily for two reasons.

The first is speed. Real-time operational flight programs like those used in flight control, navigation and weapon control systems must be able to react very quickly to changing inputs from pilots and sensors.

The second reason is memory limitation. Aircraft and spacecraft don't have access to massive servers or cloud storage. Low-level embedded code has to make do with the memory chips it has in its own box. To deal with these constraints, programmers often have to resort to non-standard coding structures to make the code run as efficiently as possible.

Having been developed to verify software in safety-critical systems, exhaustive static analysis is ideally suited to verifying low-level code.

As mentioned earlier, traditional static analysis tools are not designed to deal with these complex, non-standard structures. In contrast, exhaustive static analysis tools, having evolved in the aerospace sector, were designed to look at the code the way the hardware sees it. For example, our tool, TrustInSoft Analyzer, has specific features that enable you to specify precisely the hardware characteristics and the toolchain. Based on this configuration, TrustInSoft Analyzer sees the software product as it will be and run on the final hardware.

Sound tools

Exhaustive static analysis tools are what are called "sound" in the context of formal methods. That is, they are designed so they will not miss a single defect and can be used to guarantee that a software program is completely free of bugs and security vulnerabilities. They can even be used to guarantee that the program complies exactly with its specification.

By being just as precise as the code's compiler, these tools are able to thoroughly understand complex low-level code and perform precise mathematical analyses on it.

Also, since the formal methods employed in exhaustive static analysis are mathematical proving techniques rather than discrete test cases, they can be applied to wide ranges of input values all at once. The iterative application of different sets of input values performed in traditional software testing is not required when using formal methods. Ultimately, exhaustive static analysis saves a lot of time in verification.

Designed so any developer can use them easily

What's more, exhaustive static analysis does all this in a way that does not disrupt the normal software development process. It brings the value of mathematical formal methods to software development in a way that's practically invisible to the developer.

Within an exhaustive static analysis framework, all the aforementioned selection and application of formal methods are totally transparent to the developer. From the users' perspective, they are simply testing their code in a manner very similar to what they're already accustomed.

The framework expands and automates the testing process. For example, by adding just a couple of lines of code to your test script, you can expand a limited set of test cases into a complete set of test cases covering the complete range of possible values for all input variables.

Again, you don't need a PhD in formal methods to use these tools. Any developer can handle them. In most cases, users don't even need to be aware of the method the tool is using. They just express the problem and the tool does the analysis automatically. It's extremely easy for users to find answers. The framework chooses the right tools for the problem and switches tools on the fly.

For example, our own platform, TrustInSoft Analyzer for C/C++, employs several abstract memory models to analyze the software being validated. This process is completely hidden from users. They needn't know anything about them. TrustInSoft Analyzer has been designed and developed so that no matter how a C/C++ developer programs, it will extract the meaning of the code so that the right formal methods are applied. It was cited in a National Institute of Standards and Technology report to the White House[14] for having demonstrated that it can be used to formally guarantee that (1) no known undesired behaviors (bugs) are present in a system and that (2) the system behaves exactly according to its specification[15,16].

Ideal for verifying OS kernels

Exhaustive static analysis can also guarantee the absence of vulnerabilities or reliability critical bugs in an operating system. An OS is, after all, a bare metal program—one that provides layers of abstraction for the applications that run on top of it.

Developers of the operating systems used in many of today's connected devices are finding it prudent and justifiable to use exhaustive static analysis to guarantee their OS is defect-free. This is true for those developing operating systems for their own products (like mobile phones and gaming consoles, for example), as well as for those who license an off-the-shelf OS (like real-time OS and hypervisors) for other manufacturers to use in the devices they produce.

At the moment, we have several customers who are so concerned about the security of their devices that they have chosen to develop their (micro) OS themselves. They use TrustInSoft Analyzer to verify the security of that OS.

Clearly, the next step is for device manufacturers to demand that the off-the-shelf operating systems they are licensing from third parties are guaranteed to be free of all coding defects a hacker could exploit or critical reliability bugs. Using TrustInSoft Analyzer, OS providers can provide that guarantee today.

Two use case examples

Customers are using TrustInSoft Analyzer in a variety of ways to ensure the security of their products. Here are two examples.

Use case example #1: Red team activities

One of our customers has a red team—a team of software security experts—who audits all the low-level layers of their gaming platform.

At the end of every release cycle, they conduct a "white hat" exercise, trying to find vulnerabilities that "black hat" hackers might maliciously exploit. The exercise assumes hackers have obtained the code that is to be released. The red team performs black box penetration testing and white box analysis using TrustInSoft Analyzer.

With TrustInSoft Analyzer, they can find any subtle issues that may have slipped through the normal verification process.

Use case example #2: Continuous verification

Another customer has integrated TrustInSoft Analyzer into their continuous verification process for their trusted execution environments (TEE).

They don't wait until a release to check their code. Instead, every modified line in their code base is reanalyzed automatically with TrustInSoft Analyzer. This continuously repeating process limits the number of issues uncovered with each check and allows developers to resolve them quickly.

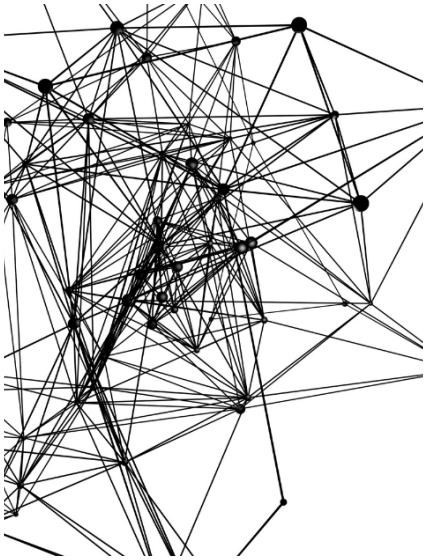
Through continuous verification, bugs and vulnerabilities are not left to accumulate over long periods of time. This helps keep the development process on track and preempts any nasty, last-minute surprises that might slip your release date.

By the way, the customer who uses TrustInSoft Analyzer in its red team activities also uses it for continuous verification.

LOW LEVEL CODE

What to look for when choosing an exhaustive static analysis solution

Choosing the exhaustive static analysis solution that's right for your organization can be challenging. Here are three important features to look for:



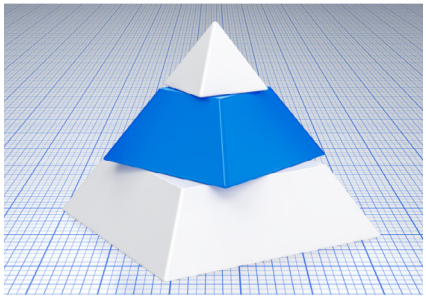
1. Applies a wide range of formal methods in a manner transparent to the user

The field of formal methods covers a variety of methods that are used to solve problems of logic and mathematics. Each method was created and is best used for a specific type of problem. A good exhaustive static analysis tool will be able to apply a wide range of these methods to solve a wide variety of problems. Only then can it hunt down the full range of possible undefined behaviors and guarantee the security and functionality of your low-level code and the data it should protect.

Furthermore, the tool should automatically select and apply specific formal methods without the need for user intervention. Scientists in the field of formal methods spend years learning how to apply these complex techniques. Your developers shouldn't have to.

2. Fits seamlessly into your current development process

Adopting an exhaustive static analysis tool shouldn't disrupt your current development process or even cause you to alter it significantly. Using the tool should be similar to the experience you now have using conventional software testing tools.



3.Offers several hierarchical levels of analysis

Tracking down and eliminating undefined behaviors and guaranteeing the security and reliability of low-level code is an iterative, cumulative process. The process benefits from taking a step-by-step hierarchical approach, stepping from a basic level of proof to more advanced levels, depending on what a given application requires.

For example, TrustInSoft Analyzer offers three levels of analysis.

Level 1 (fast analysis) is the simplest to use. It is completely automated—as easy as ordering a compile of your code. The user doesn't even have to look at the code to use it. Yet, Level 1 will uncover a large portion of the bugs present in the code without yielding any false positives.

Level 1 is ideal for use in a continuous integration process. New bugs can be stripped out on a daily basis before they accumulate, without developers having to look for them.

Level 2 (exhaustive analysis) will guarantee that all coding defects have been eliminated from your code—that you have no undefined behaviors present that can be exploited by hackers. Use of Level 2 requires some operator intervention, but it has a very low false positive rate (<10/10,000LOC on average) and no false negatives. Level 2 provides you with a guarantee of the security of your system if all the confirmed defects are corrected.

Finally, for those applications that need it, there is Level 3 (functional proof). Level 3 guarantees your code fulfills its specification exactly. Functional proof takes longer and is more costly than defect testing, but for applications that need to be perfect—OS kernels, hypervisors and certain firmware applications like secure boot, for example—it is well worth it. The return is huge. You have a guarantee that your critical application works exactly as specified.

Conclusions

Embedded code is everywhere. Its security is pivotal to the well-functioning of our society, whether in critical systems or consumer electronics. Connected device manufacturers, their corporate customers, and consumers all need and expect their devices to protect their valuable, sensitive data from theft or destruction by malicious hackers.

In parallel with software security, software reliability plays a key part in software quality. Software reliability is difficult to achieve. As more and more software is embedded into systems, all stakeholders need to be confident that the software will not cause any disasters.

Along with electronic hardware, low-level software and firmware are the bedrock of trust for these devices. Unfortunately, traditional static analysis tools and software testing are no longer adequate for ensuring high-quality code and the necessary level of code security and reliability in low-level applications.

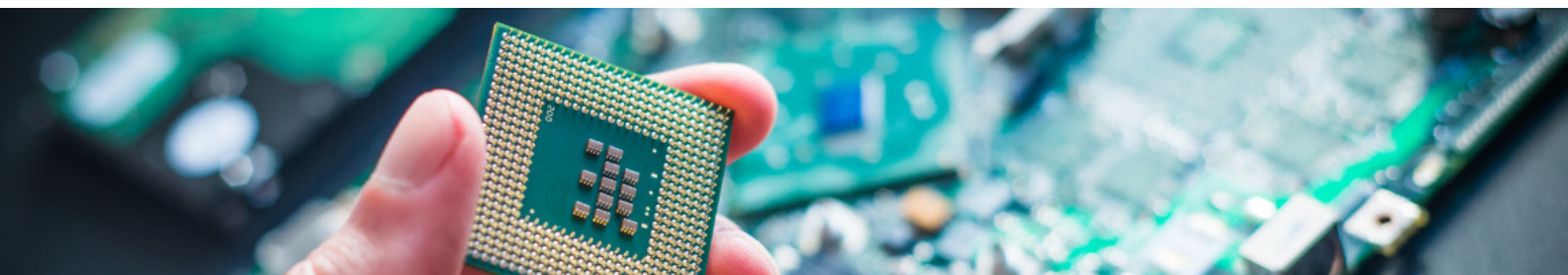
Traditional static analysis tools, having been designed for use on top-layer applications that run on operating systems, are not well-adapted to analyzing intricate low-level code. Traditional software testing is too time-consuming and therefore too costly to handle the scope of the problem. The number of test cases and possible input combinations is far too great.

Exhaustive static analysis, based on mathematical formal methods, is a solution to the principal challenges of verification and validation in embedded environments. It can guarantee the total elimination of the vulnerabilities from low-level code that hackers exploit to breach embedded systems and other electronic devices, as well as the coding errors that compromise their reliability.

LOW LEVEL CODE

References

1. Hasan, Mohammad; ***State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally***, IoT Analytics, May 2022.
2. ***Security Signals***, Microsoft, March 2001.
3. Seals, Tara, ***80% of Global Enterprises Report Firmware Cyberattacks***, Threatpost, April 2021.
4. ***The Top Five Firmware Attack Vectors***, Eclypsium, December 2020.
5. Joshi, Praveen, ***Common Attacks On Embedded Systems And How To Prevent Them***, RSK Cyber Security, August 2022.
6. Ibid.
7. ***2019 CWE Top 25 Most Dangerous Software Errors***, Mitre, July 2021.
8. Joshi, Praveen, ***Common Attacks On Embedded Systems And How To Prevent Them***, RSK Cyber Security, August 2022.
9. Pieter Arntz, ***Critical WhatsApp vulnerabilities patched: Check you've updated!***,Malwarebytes Labs, September 2022.
10. Dunn, Michael, ***Toyota's killer firmware: Bad design and its consequences***, EDN, October 2013.
11. ***ICS Advisory (ICSMA-17-250-02A): Smiths Medical Medfusion 4000 Wireless Syringe Infusion Pump Vulnerabilities (Update A)***, CISA, September 2017.
12. Goodin, Daniel, ***Boeing 787 Dreamliners contain a potentially catastrophic software bug***, Ars Technica, May 2015.
13. Moy, Y., Ledinot, E., Delseny, H., Wiels, V., Monate, B., ***Testing or Formal Verification: DO-178C Alternatives and Industrial Experience***, IEEE, April 2013.
14. Black, P., Badger, L., Guttman, B., Fong, E., ***Dramatically Reducing Software Vulnerabilities: Report to the White House Office of Science and Technology Policy; National Institute of Science and Technology (NIST)***, November 2016.
15. Bakker, P., ***Providing assurance and trust in PolarSSL***, Offspark, May 2014.
16. Regehr, J.; ***Comments on a Formal Verification of PolarSSL; https://blog.regehr.org***, September 2015.



About TrustInSoft

Since our beginnings, TrustInSoft Analyzer has been adopted by industry-leading companies around the world to ensure sound cybersecurity in their low-level code.

TrustInSoft participates in the Application Security Testing market alongside vendors such as Mathworks, Parasoft, Synopsys and Veracode. TrustInSoft Analyzer is a hybrid static and dynamic code analyzer that automates Formal Methods to mathematically guarantee C/C++ code quality, security and safety. TrustInSoft has customers worldwide in the automotive, IoT, telecom, semiconductor, aeronautics and defense industries. The company received awards and recognition from NIST, RSA and Linux Foundation.

To learn more about TrustInSoft Analyzer, visit trust-in-soft.com/product/trustinsoft-analyzer.

If you'd like to speak with a TrustInSoft technical representative about how TrustInSoft Analyzer can meet your organization's specific needs, contact us by email at contact@trust-in-soft.com.

TRUST IN SOFT

contact@trust-in-soft.com
Phone : +33 1 84 06 43 91 or +1 (408) 829-5882