Editors: Eric Bodden, eric.bodden@uni-paderborn.de | Fabio Massacci, fabio.massacci@ieee.org | Antonino Sabetta, antonino.sabetta@sap.com

How to Solve Cybersecurity Once and For All

Marcel Böhme | Max Planck Institute for Security and Privacy

At last year's Pwn2Own competition, one individual successfully exploited all major browsers—Chrome, Firefox, Safari, and Edge—used by billions of people worldwide. Despite decades of security research, the discovery of new vulnerabilities in important software systems continues unabated.

Building security into software from the start is the most effective approach to cybersecurity. Unlike physical systems, where behavior is studied empirically, software systems are fully described through source code, which reflects the programmer's intentions using the syntactic and semantic rules of the programming language. Because software operates based on well-defined instructions, we can theoretically reason about, control, and monitor its behavior with great precision. By developing increasingly better security tools and processes, in the limit, we should be able to prevent attackers from launching successful exploits. Is this how we can solve cybersecurity once and for all?

How to Solve Cybersecurity Once and For All

Imagine we have used all available tools and processes to design, develop, and maintain our software system with security as first-class citizen.[1] We've applied offensive and defensive strategies to find and fix flaws, created threat models, and adopted best practices, like using memory-safe languages and rigorous secure software engineering principles. We also run continuous testing, such as fuzzing and security tools (static/dynamic application security testing, SAST/DAST), and even formally verify critical components. But is this enough? Are we truly safe?

Is it possible for a software system to be completely free of security flaws? If not, why bother?

Now, imagine you're the vendor of a widely used mobile phone. Despite your best efforts to protect security and privacy, the first jailbreak is released within two weeks. After patching it, a new jailbreak appears just months later. Even after extensive work to secure everything, new jailbreaks keep appearing. Over the next two decades, you invent critical mitigations, many of which have been adopted as de facto industry standard, only to see the next jailbreak finally trigger another security update. Does this mean that your defenses are ineffective? Definitely not.

No Universal Claims About Security

There are at least two reasons why we cannot guarantee for any software system that it is free of security flaws. First, there are the unknown unknowns: We don't know what we don't know. For a system to withstand attacks, we must know which properties must hold. In many cases, we only know that some software behavior is actually a security flaw retrospectively. For instance, speculative execution—a performance optimization technique where processors predict and execute instructions before knowing if they are actually needed—was meant to improve the performance of our processors, and it does in almost all cases. However, it took someone with a security perspective and a decent amount of curiosity to find that we require all secret-dependent executions (e.g., in a cryptographic protocol) to run in constant time: Meaning they must take exactly the same amount of time regardless of what secret values are being processed. This constant time property is violated by speculative execution. An attacker could measure subtle timing differences to infer the secret values, effectively breaking the cryptographic protection. Now, how do we validate or enforce this high-level

Digital Object Identifier 10.1109/MSEC.2025.3551590

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

Copublished by the IEEE Computer and Reliability Societies

constant-time property in the concrete? Do we disable speculative execution (Spectre)? Do we count (fast) cache hits and (slow) cache misses (MeltDown)? Do we disable all data-dependent optimization in the compiler and processor? Whatever we choose, the validation or enforcement of the high-level property is always specific. So, how can the defense (e.g., against attacks on constant-time) ever be general? I claim that an attacker can always exploit: 1) the absence of properties that we do not even know need to hold and 2) the specificity with which we must validate or enforce that high-level property.

Second, there is the modeling gap: For efficiency, we usually reason within some model of the behaviors of a system; and from properties of the model, we make claims about the actual, deployed system as it is running in production. For instance, provable security—including cryptography, model checking, protocol analysis, secure-by-construction, proof-carrying code, and software verification—models the full behavior of the system using formal methods, and it allows universal statements about the system's properties. SAST techniques—like symbolic execution, abstract interpretation, and logic-based static analysis, including Infer, CodeQL, SonarQube, and FindBugs—model all relevant executions of a program after parsing its source code or binary to automatically check assertions about behaviors of the system. DAST techniques—like sanitization, compartmentalization, sandboxing, and trusted execution environments—model the current execution at some (fixed) level of abstraction. Approaches from secure-by-design—like security best practices, threat modeling, and language-based security—model future software systems before they are built and avoid introducing security flaws at the design stage.

An attacker can always exploit an invalid assumption about or the higher level of abstraction of the actual system. For instance, imagine running a formally verified Rust implementation of a provably-secure protocol on a fleet of CPUs with a microcode bug. In 2023, a vulnerability (CVE-2023–20593[2]) was found in AMD Zen 2 class processors that would leak the parameters of security-critical basic operations, like `memcpy` or `strcmp`, in plain text, across the boundaries of virtual machines, sandboxes, containers, and processes. You only needed to trigger an XMM Register Merge Optimization followed by a register rename and a mispredicted vzeroupper within a precise window of time. Security guarantees established at the protocol-level, the source-code-level, or even for the executable fail to hold if the processor does not do what we assume it to do. In the gap between source code and executable, we can find undefined behavior, which causes 72%[3] of exploits in-the-wild, 86%[4] critical vulnerabilities in Android, and 70%[5] in Chrome. In the gap between the executable and the process running on the machine, we find hardware-specific vulnerabilities, such as the microcode bugs, side channels, and RowHammer (a hardware vulnerability where repeatedly accessing certain memory rows in DRAM chips can cause bit flips in adjacent rows, allowing attackers to alter memory they should not have access to and potentially gain unauthorized privileges). Given only the program, without assumptions about the compiler or the machine, it is hard to make reliable statements about properties of the running process.

Secure or Insecure. That Is Not the Question

Let's get back to our earlier question: If there are no guarantees, why bother? Well, the security of a software system is no binary property that needs to be guaranteed at all. In practice, security is fundamentally empirical and more like a numeric property that needs to be strengthened. Hence, the true purpose of security tooling [including that which is designed to formally guarantee security (in the specific)], is to increase a system's security (in the general). In other words, we want to reduce the likelihood that an attacker can compromise the security of the system, including confidentiality, integrity, or availability. If you ever find yourself considering your enterprise system as guaranteed secure, it is only because you are missing a counterexample. We can only try to approach absolute security in the limit.

We should focus on the degree to which our systems are secure. First, we should think about security as an attacker cost. In our motivating example, we talked about a phone vendor that has been reacting to jailbreaks by developing new mitigations. Some defenses were long-term, ground-breaking, and general mitigations, and others were just short-term patches, but together they ultimately rendered the cost of the next jailbreak just impractically high. While a kernel-read/write was sufficient a few years ago, it is only the starting point for a year-long journey today. Second, we should think about security in economic terms as a function of incentive. Like in the phone vendor's case, there is a huge demand for jailbreaks and thus to overcome the security measures of the vendor. On the one hand, we have the vendor's supply of security measures. On the other hand, we have the jailbreakers' demand to overcome these security measures. When there is no demand, our system may only seem secure to us and to the outside, irrespective of our supply (i.e., the strength of our defenses). If instead there is substantial demand, like in

our jailbreak example, our system may seem insecure only to the outside but not to us. I would argue that reports of successful attacks, whether through bug bounty programs or red team exercises (which increase demand artificially), or through other ethical means, provide the only reliable signal about the current strength of our defenses. Interestingly, this also means that a system with a larger number of publicly known security flaws (common vulnerabilities and exposures, or CVEs) may technically be much more secure than a system with no CVEs, at all.

Consequently, if our tools and processes fail to find or prevent a specific security flaw, even if the reader works in software verification—Worry not! Our tools and processes are not ineffective; they just need to be hardened. We should identify the particular reason why they failed in that specific case and fix it. We strengthen our security defenses one reported security flaw at a time, entirely incrementally and in a counterexample-guided manner. Since security is an empirical property, we must take an empirical approach. Instead of claims about the effectiveness of our defenses, we should consider focusing on claims about our failure to find counterexamples.

Security Engineering

The real strength of security tooling is measured by the failure of those with enough incentive to find successful attacks. Still, we evaluate our scientific progress by confirming known attacks to be unsuccessful. Instead, we should focus more on identifying security flaws that our tools cannot find or mitigate. For instance, as developers (or users) of a static analysis tool, we should evaluate SAST performance with a focus on those security flaws clearly in scope, that it fails to find and elicit the underlying reasons as limitations for future research. Just confirming that our technique worked for a larger number of security flaws in a specific benchmark will not help to inform our scientific progress. The focus moves from universal claims about the security of a software system to falsification of such universal claims. Just going from 99% to 100% on some benchmark tells us nothing about their failure to find or mitigate future attacks.

We might agree that nothing can truly guarantee security. Yet, we perpetually develop new techniques for every new type of vulnerability that is currently undiscovered by existing means. Given that there will always remain some insecurity, new techniques might represent no progress at all. Instead, for every attack that is successful despite the defenses we employ, we should ask ourselves what exactly caused this failure in our defenses, and what exactly can be improved in our defenses to find or mitigate the most general version of the attack in the future. In this way, our defenses can empirically "converge" toward a fixed point where no counterexamples can be found within reasonable cost. In fact, existing tools always fail to find a given (type of) vulnerability for a specific reason. Why not localize and address exactly that reason? This counterexample-guided hardening perspective offers a longer-term approach where existing techniques are systematically extended rather than eternally complemented.

In the absence of successful attacks to use as counterexamples for our defenses, we can artificially increase the "demand" (i.e., the incentive to report successful attacks) and thus turn our reactive approach into a proactive one. For instance, using an effective bug bounty program, we invite bug reports and also receive some signal on the strength of our defenses. In the absence of successful attacks, we know that the software system is at least as secure as our bug bounty program is willing to pay for evidence of insecurity.

Hardening our Software and Defenses

How do we solve cybersecurity once and for all? There are no guarantees in security (there may always be an unknown attack that could be successful), but we can at least approach maximal security in the limit in a counterexample-guided manner. Consider fishing as a metaphor for bug finding. A *fishernet* represents the tools and processes we use to build security in while the *fishes* represent the security flaws in our software systems. My claim is that, for every net, there will always be a fish that slips through. But clearly, this does not undermine the utility of the net. We must realize that there is not ever going to be the ultimate fishernet. Rather, I have argued, we should develop more systematic support for an incremental, counterexample-guided evolution of our defenses to maximize effectiveness empirically. Our focus moves from developing new defenses conceptually to identifying and mitigating the limitations of our existing defenses empirically. But what does it mean?

For the defensive security community this means that, when evaluating a new technique, instead of evidence supporting our claims about its effectiveness, we should actively seek evidence that could falsify them. At least, we should add discussions or evaluations of the risk of potential attacks despite that new technique. Indeed, we should design our techniques with the possibility to render it effective, reactively, against currently unknown types of attacks (e.g., like CodeQL allows adding new detection rules).

For the offensive security community this means that any new type of attack effectively becomes a counterexample for all of our existing

defenses. Hence, in addition to the analysis of the attack itself, we should also add an analysis of why existing defenses fail to catch this attack. We should provide concrete advice on how these existing defenses can be hardened against the most general version of this attack.

For the software engineering community this means that there are opportunities to automate this hardening process using techniques from automated software engineering.[6,7] We can develop techniques to automate the localization of the root cause of a failure in our defenses given a successful attack (as in automated debugging) and further to automatically render our tools effective against the new attack (as in automated program repair).

For the general reader of this article this means that we should stop trying to confirm the effectiveness of our defenses and start failing to find counterexamples to their effectiveness. This is how we solve cybersecurity once and for all. One counterexample at a time. ■

Acknowledgment

This article summarizes an invited keynote address at the 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024) with the same title.

References

1. M. Böhme, E. Bodden, T. Bultan, C. Cadar, Y. Liu, and G. Scanniello, "Software security analysis in 2030 and beyond: A research roadmap," *ACM Trans. Software Eng. Method.*, pp. 1–24, 2025, doi: 10.1145/3708533.
2. T. Ormandy, "Zenbleed." cmpxchg8b.com. Accessed: Jul. 24, 2023. [Online]. Available: https://lock.cmpxchg8b.com/zenbleed.html
3. "0day in the wild," *Google Project Zero*, Apr. 20, 2023. [Online]. Available: https://docs.google.com/spreadsheets/d/1lkNJ0uQwbeC1ZTRrxdtuPLCIl7mlUreoKfSIgajnSyY/edit
4. "Memory safe languages in Android 13," *Google Android Secur.*, Dec. 1, 2022. [Online]. Available: https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html
5. "Memory safety," Google Chromium Secur., 2022. [Online]. Available: https://chromium.org/Home/chromium-security/memory-safety/
6. A. Zeller, "The debugging book," CISPA Helmholtz Center for Inf. Secur., Saarbrücken, Germany, 2024. [Online]. Available: https://www.debuggingbook.org/
7. C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, Nov. 2019, doi: 10.1145/3318162.

Marcel Böhme is a faculty member at the Max Planck Institute for Security and Privacy, 44799 Bochum, Germany, where he leads the Software Security research group. His research interests include foundations of cybersecurity, fuzzing (automated testing), and ML4Sec. Böhme received a Ph.D. from the National University of Singapore. He won a 2024 European Research Council Consolidator grant and serves as a guest editor in chief and associate editor of *ACM Transactions on Software Engineering and Methodology*, and as a Program Committee chair of the Association for Computing Machinery (ACM)/IEEE Automated Software Engineering Conference 2025 and ACM SIGSOFT International Symposium on Software Testing and Analysis 2026. Contact him at marcel.boehme@acm.org.