Fundamental Challenges in Cybersecurity and a Philosophy of Vulnerability-Guided Hardening

Marcel Böhme
MPI-SP, Germany

***Abstract*—Research in cybersecurity may seem reactive, specific, ephemeral, and indeed ineffective. Despite decades of innovation in defense, even the most critical software systems turn out to be vulnerable to attacks. Time and again. Offense and defense forever on repeat. Even provable security, meant to provide an indubitable guarantee of security, does not stop attackers from finding security flaws. As we reflect on our achievements, we are left wondering: Can security be solved once and for all?**

In this paper, we take a philosophical perspective and develop the first theory of cybersecurity that explains what *fundamentally* prevents us from making reliable statements about the security of a software system. We substantiate each argument by demonstrating how the corresponding challenge is routinely exploited to attack a system despite credible assurances about the absence of security flaws. To make meaningful progress in the presence of these challenges, we introduce a philosophy of cybersecurity.

I. Introduction

In the Pwn2Own hacking contest this year, a single person (Manfred Paul) demonstrated successful exploits for all major browsers: Google Chrome, Mozilla Firefox, Apple Safari, and Microsoft Edge [1]. Just Chrome is used by 3.5 *billion* people.[1] Today, thanks to Paul, these major security flaws are fixed. Last year, evidently, most people accessed the internet with the door open to potential attackers—despite decades of research in defensive security, despite an abundance of mitigations, despite a dedicated red team (Google P0) that is considered the best in the world. From the outside, research in cybersecurity might seem technology-specific, reactive, and ostensibly ineffective.

But why? Software is entirely virtual and can be described completely [2]: A program's source code is meant to formally express the programmer's intention using the syntactic and semantic rules of the programming language. As the behavior of a software system arises from well-defined instructions, we must be able to formally reason about all its properties. Surely there exists an approach that will forever guarantee the security of our systems. Only, we have not found it, yet?

> *Fundamentally, can we guarantee the absence of security flaws? If not, what precisely prevents us from making reliable statements about the security of a software system?*

This question about the effectiveness of program analysis is at the very heart of *Software Engineering*. Yet, our battle-tested research methods are not well equipped to answer it. Since we do not propose a new approach or intervention, we can hardly run measurements or experiments to gain insight. Since we look for guidance on the science of security itself, we cannot expect answers from descriptive methods, either.

[1] https://backlinko.com/chrome-users

In this paper, we take an objective, philosophical perspective on the effectiveness of cybersecurity tooling and develop a *theory of cybersecurity* that explains why we must, in general, assume that exploitable security flaws exist in any sufficiently large software system. We substantiate each argument by demonstrating how the corresponding challenge is routinely exploited to attack a system despite credible assurances about the absence of security flaws. We believe, now is the time to reflect on our scientific progress more fundamentally and to shed light on the limitations of our approaches to reason about the properties of a system. We do not claim, our theory is final or complete. Instead, we expect it to be discussed, improved, and refined in a protracted scientific debate.

How do we make meaningful progress? The quest for truth in the absence of a mechanism guaranteed to arrive at the truth has existed since the time of enlightment. In his philosophy of science, Popper suggests, scientific theories evolve in a counterexample-guided manner (*conjectures and refutations*) [3]. In his philosophy of maths, Lakatos suggests, even theorems evolve in a counterexample-guided way (*proofs and refutations*) [4]. Similarly, in our philosophy of security, we suggest, security claims evolve via counterexamples, too.

Consider fishing as a metaphor for bug finding. A *fishernet* represents our tools and processes while the *fishes* represent the security flaws in our software systems. Our theory explains why, for every net, there will always be a fish that slips through. Clearly, this does not undermine the utility of the net. Our philosophy turns the objective from developing the ultimate fishernet to an incremental, counterexample-guided evolution of our nets maximize effectiveness empirically.

Concretely, our philosophy suggests a *vulnerability-guided hardening* of our security approaches. We cannot assume that an approach T developed to protect software system S is final in any way. Instead, we recommend (i) to find some evidence c *against* the security of S despite T (e.g., by exploiting the challenges we identified), (ii) to patch S and T so as to account for c, and (iii) to restart by finding the next evidence c'. Later, we discuss a concrete application to software verification.

Like counterexample-guided approaches in science and maths, vulnerability-guided hardening addresses the identified fundamental limits *empirically*. In contrast to science and mathematics, we can support this method with automation. For instance, we should invent novel debugging techniques for our approaches T that allow us to pinpoint the precise root cause of their failure to discover or mitigate a given vulnerability c. We need innovative techniques to programmatically reconcile the discovery of a vulnerability c to improve the approach T.

In summary, this paper makes the following contributions:

- We introduce a *theory of cybersecurity* which explains why we must assume that exploitable security flaws exist in any sufficiently large software system. We discuss nine (9) fundamental challenges for software security analysis and demonstrate how they are routinely exploited.
- We introduce a *philosophy of cybersecurity* which serves as guiding principle granting the fundamental insecurity of our software systems.[2] We suggest that a security mechanism cannot be deemed effective *generally* and *with finality*, e.g., if it performs well on a benchmark. Instead, we must incrementally find evidence *against* the generality of our security mechanisms by subjecting them to a method of systematic scrutiny, we call vulnerability-guided hardening.
- We introduce *meta verification* to instantiate our method of vulnerability-guided hardening for software verification and to tackle the long-standing crux that formal guarantees for empirical systems are empirically unreliable [5]–[7].

II. Background

A. Science of Security

In pursuit of more rigorous foundations, there have been many calls for making the research discipline of cybersecurity more scientific [8]–[12]. For instance, Schneider [13] argues that the cybersecurity research community ought to construct a body of laws for predicting the consequences of design and implementation choices. The laws should "(i) transcend specific technologies and attacks, yet still be applicable in real settings, (ii) introduce new models and abstractions, thereby bringing pedagogical value besides predictive power, and (iii) facilitate discovery of new defenses and describe non-obvious connections between attacks, defenses, and policies, thus providing a better understanding of the landscape".

Herley and Van Oorschot [5] provide an excellent survey of such calls for a more scientific approach and ponder what it means for security to be "more scientific". They start with a comprehensive survey of view points in Philosophy of Science and discuss concrete opportunities for cybersecurity generally.[3] On our topic of mechanisms for reliable statements about the security of a software system, they discuss *provable security* specifically, i.e., techniques that use the tools of logic, formal methods, cryptography, and mathematics to derive a formal guarantee about the security of a system.[4]

Provable security is *not technically* a science in the same way as mathematics is not technically a science, the authors argue [5]. While a *science* seeks to make inductive statements from observations about the empirical world (which requires hypotheses to be falsifiable), *provable security* seeks to make deductive statements from axioms within a formal system. This observation is further explored by Murray and Van Oorschot [7] who expose challenges of interpreting the formal guarantee for the actual system (which is labeled as "formally verified") and of enforcing the formal assumptions.

Herley and friend [5] call this the *inductive-deductive split*: "Speaking of mathematical guarantees as if they are properties of real-world systems is a common error. [..] It is worth being unequivocal on this point. There is no possibility whatsoever of proving rigorously that a real-world system is [..] invulnerable to (all) attacks" [5]. In this paper, we make this assertion more precise by identifying concrete challenges (beyond provable security), but also fundamentally resolve this "inductive-deductive split" which Herley and Van Oorschot consider as insurmountable. We agree that formal claims are not guaranteed to hold for the empirical system but suggest that the distance between the formal and the empirical world is iteratively reduced via (empirical) counterexamples by applying the scientific method to the *process* of provable security itself (inspired by Lakatos' philosophy of mathematics [4]).

While there exist many approaches to analyse the security of a software system, only provable security provides a *formal guarantee*. The concrete assumptions, the concrete properties, and the formal reasoning framework are explicitly, precisely, and formally spelt out. A proof gives a formal and universal guarantee that *these* properties hold within *this* model of the software system if *these* assumptions are met.

Koblitz and Menezes [14]–[16] critizise the finality with which these guarantees are advertised. They provide examples of incorrect proofs, non-constructible algorithms, idealized models, broken assumptions, and inadequate definitions in cryptography. Indeed, Lakatos [4], [17] argues that proofs of mathematical theorems aren't final, either. Rather, he suggests, these water-tight deductions from well-defined premises are the perhaps temporary end-points of an evolutionary, dialectical process. De Millo et al. [6] observe that mathematics evolves within a *social process* where the mathematicians' confidence in the validity of a theorem increases as it becomes subject to more scrutiny; they assert that software verification is "bound to fail" precisely because of the lack of this social process. In this paper, we build on Lakatos' philosophy of mathematics to develop a philosophy of security. We explain how De Millo's social process may not be required, after all, and suggest how it may be substituted by a general, systematic counterexample-guided method.

> *"The aim of program verification [..] is to increase dramatically one's confidence in the correct functioning of a piece of software. [..] Contrary to what its name suggests, a proof is only one step in the direction of confidence. We believe that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem–and we believe that, because no comparable social process can take place among program verifiers, program verification is bound to fail."*
>
> —De Millo, Lipton, and Perlis [6]

[2] To clarify, a philosophy is really just a specific way of thinking about a problem. It represents a particular perspective on the nature of that problem.

[3] Herley and Van Oorschot [5] also critize how empirical work in security aims to *verify* existing beliefs while it really should falsify them to disambiguiate possibilities and suggest new theories. Furthermore, referring to defensive advice like "effective passwords must have a certain entropy", they criticize the advancement of unfalsifiable claims and the conflation of unsupported assertions and argument-by-authority with evidence-supported statements.

[4] Including cryptography, model checking, software verification, verified compilation, proof-carrying code, and (exhaustive) symbolic execution.

B. Generating Reliable Statements about Software Security

1) What is a secure software system? An attacker should not be able to read what they shouldn't (*confidentiality*), to write what they shouldn't (*integrity*), or to disturb the service of others (*availability*).

2) Attacks. Today, the largest class of security flaws is due to *undefined behavior* and *memory unsafety* more specifically. This includes buffer overflows, heap/stack smashing, return-oriented programming (ROP), return2libc, and format-string vulnerabilities. Other attacks exploit *data races*, including time-of-check time-of-use (TOCTOU) vulnerabilities, or *hardware defects* such as RowHammer [18].

Several types of attacks are related to problematic information flow. An *injection attack* is a problematic information flow from a public source to a sensitive sink that might threaten integrity. This includes command injection, code injection, SQL injection, Log4Shell (JNDI) [19], Cross Site Scripting (XSS), and deserialization vulnerabilities. An *information leakage* is a problematic information flow from a sensitive source to a public sink that might threaten confidentiality. This includes hardware and software side channel vulnerabilities [20].

Many attacks are domain-specific. There are hypocrite commit vulnerabilities and compiler backdoors for in *supply chain security*. There are man-in-the-middle (MitM) and spoofing attacks in *network security*. There are click jacking and XSS attacks in *browser security*. There are jailbreaks and communication-stack-based attacks in *mobile security*. There is privilege escalation in *OS and hypervisor security*.

3) Defenses. The best defense is *education and training*. To make our systems more secure, we should teach developers adversarial thinking [21] as well as offensive and defensive strategies in hands-on labs and capture-the-flag (CTF)-type competitions. At the university-level, students should be able to take a long-term, technology-unspecific, automation-centric perspective on software security in-the-large.

Defenses (or mitigations) require us to draw a line between what an attacker *must never* be allowed to do (read, write, or disturb) and what the user *must always* be allowed to do. For instance, the *principle of execution integrity* suggests to establish programmer's intention and to avoid deviations. For memory safety, this includes control-flow integrity (CFI) where the intended control-flow is statically precomputed and dynamically enforced, but also canaries, shadow stacks, and fat pointers. Language-based security requires intention to be made explicit, e.g., via borrow checking.

The *principles of isolation* and *least privilege* require us to specify isolation primitives and authorization policies. For memory unsafety, memory is segmented and guard pages introduce, components are compartmentalized, and pointers signed. For data races, concurrent processes are isolated using locks and semaphors. For injection attacks, sensitive sources are isolated from public sinks. For information leaks, public sources are isolated from sensitive sinks. For attacks on operating systems, processes are assigned to rings or sandboxes, and privileges / permissions are assigned to users.

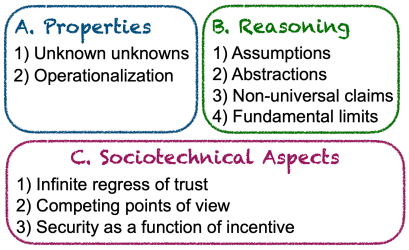

Fig. 1: Overview of the identified challenges to make reliable statements about the security of a software system.

III. A Theory of Guarantees in Cybersecurity

We develop a first theory that explains why we must, in general, assume that security flaws exist in any sufficiently large software system. Our theory elucidates specific blindspots in our approaches to reason about security and how they can be exploited by an attacker. As evidence for our theory, for each blindspot, we discuss concrete examples where that blindspot was exploited, often despite credible assurances about the absence of security flaws. We distinguish these challenges in three categories (cf. Fig. 1): (A) the *security properties* we need to establish and operationally define, (B) the *reasoning system* (or security mechanism) that we use to check or enforce these properties, and (C) the *sociotechnical challenges*.

A. Challenges to Distinguish Secure from Insecure Software

We cannot perfectly recognize and precisely describe the distinction between secure and insecure software.

On the one hand, we agree that an attacker should *not* be able

- to read what they should not read (confidentiality),
- to write what they should not write (integrity), or
- to disturb the service for others (availability).

For instance, a cryptographic system whose execution time depends on the secret key violates the confidentiality criterion. For the cryptographic system to be secure, the constant-time property must hold (amongst other properties).

Drawing the line. On the other hand, we may not recognize *all* properties that need to hold for the system to be secure (unknown unknowns). Even if we did, we may *not* be able to describe those properties precisely enough to detect, prevent, or mitigate *all instances* of their violation (operationalization).

1) Which line? Unknown unknowns. The first fundamental challenge is to know *which* properties should need to hold for a system to be secure. We may be absolutely convinced that no security flaws exist—until some behavior, that is problematic only in retrospect, is exploited by an attacker. We call such "unknown unknowns" as *black swan properties*.

Example 1. For instance, Nakamoto's consensus protocol has been formally shown to *"guarantee safety and liveness"* [22]. However, there is nothing preventing malicious participants from violating their part of that protocol and casting their vote more than once. To account for this, we need a secure consensus protocol to have another property called *accountable safety* [23]. Moreover, there is nothing preventing

an attacker from creating network partitions such that the block confirmation procedure cannot be relied upon. To guarantee safety under network partitions, we need a secure consensus protocol to have another property called *finality* [24]. Many other properties need to be satisfied for a consensus protocol to be able to withstand attacks, some of which are known while others remain unknown until their absence is exploited.

Example 2. In 2018, Google introduced Android 9 Pie with a tool called Markup that allowed users to crop images. In 2023, Security researchers found that this crop is only virtual (CVE-2023-21036). The cropped part of the image is only *marked* as cropped. The data remains. This violates confideniality because the user might have shared an image where sensitive information was cropped, e.g., on an image of a credit card. Today, we know that Markup should ensure that cropping *actually* removes the cropped image data.

Example 3. Eleven years ago, Murray et al. [25] gave the first proof of information flow security for an industrial-strength, high-performance operating system microkernel, called seL4. The formal demonstration of confidentiality and integrity with respect to the given security policy is considered a milestone in software verification. SeL4 was written in 8.7k lines of code and verified with 200k lines of proof. For this milestone, the team received the well-deserved 2022 ACM Software System award. Four years later, Spectre (CVE-2017-5753) was shown to violate the confidentiality guarantee. The speculative execution of secret-dependent branches yields small differences in execution timing which allows an attacker to infer the secret from observations of the execution time. Since then we know that seL4 must also satisfy the constant-time property to guarantee confidentiality.

Mitigation (threat modeling). Some program behaviors are security flaws only *in hindsight*. What exactly makes a system vulnerable to attacks may be difficult to anticipate beforehand. We can attempt to anticipate certain kinds of properties by following a systematic elicitation process. For instance, we can define a *threat model*, i.e., a structured representation of all the information that affects the security of an application. Some attacks will be considered within the threat model while others will be considered outside. However, some attacks will not be considered by the threat model in the first place (as our examples demonstrate).

2) Drawing the line (operationalization). Even if we know which high-level property should hold, we must define that property concretely and operationally to be able to mechanically check or enforce it. An attacker can violate a high-level property while keeping its operationalization in tact.

Example 4. We know that seL4 must satisfy the constant-time property to guarantee confidentiality from Example 3. How do we formalize this property? We could enforce that there are no secret-dependent branches that are speculatively executed to mitigate Spectre (CVE-2017-5753). We could enforce that the number of cache hits and misses is not secret-dependent to mitigate Meltdown (CVE-2017-5754). However, this is only a subset possible secret-dependent optimizations that need to be handled, today and in the future [26]. To recover the formal guarantee of confidentiality, the seL4 project would need to formalize *all* of these operational definitions.

Example 5. To prevent an attacker from executing arbitrary code, we could blocklist specific system calls that are known to directly or indirectly facilitate arbitrary code execution.[5] However, this could also block important system calls that are required for many programs to function properly, and we might forget to block other system calls that could still be exploited. In fact, there are many other ways to gain arbitrary code execution instead of system calls. We must decide concretely how we operationally "encode" that attackers should not be able to execute arbitrary code. Bypassing our operationalization, an attacker may still execute arbitrary code.

Drawing the right line. Many types of security flaws require an explicit decision between what an attacker *must never* be able to do versus what a user *must always* be able do. For instance, *injection vulnerabilities*, such as command/SQL/XSS injection (Log4Shell [19]) or deserialization vulnerabilities require input sanitization (e.g., via regular expressions), block- or allow-lists, or a list of sensitive sinks. Similarly, *information leakage vulnerabilities* (e.g., a credit card number leaking into a shared log) require lists of sensitive sources and public sinks. Because we always need to settle on a *specific* operational definition, an attacker may be able to violate the high-level property while keeping the operationalization in tact.

Using the right pen. The process of operationalization is limited by the *language* (or theory) available to encode that property. For instance, *pattern-based static analysis*[6] cannot find security flaws that cannot be expressed by the pattern language (e.g., as extended regular expression).[7] *Semantics-based static analysis* cannot find security flaws inexpressible as assertions on the modelled program state. For instance, symbolic execution[8] requires assertions expressed in the given Satisfiability Modulo Theory (SMT). Logics-based analysis[9] requires assertions expressed in the given logic (e.g., separation logic). Software verification[10] requires assertions expressed in the given theory (e.g., Linear Temporal Logic (LTL)). Constructs like pointer or floating point arithmetic, stack/heap, or string/array operations need to be modelled explicitly. In fact, papers introducing side channel attacks often come with their own logic to encode such type of attacks [30], [31].

Problem of Translation [4]. How do we know the terms *inside* the operational definition have the same meaning as the terms *outside*? In philosophy, this is the Problem of Translation. For instance, how do we know that *your* definitions of "bias" and "fairness" are equivalent to opertional definitions that researchers have developed in machine learning [32]?

[5]For instance, FlawFinder (https://dwheeler.com/flawfinder) is used to audit calls to strcpy, strcat, sprintf, chown, mktemp, exec*, system, and popen.

[6]A *static analysis* finds bugs *without* executing the program by analyzing the source code or compiled binary.

[7]Weggli https://github.com/weggli-rs, SemGrep https://github.com/semgrep, FlawFinder https://dwheeler.com/flawfinder/.

[8]Klee https://klee-se.org/, Java Pathfinder https://github.com/javapathfinder.

[9]Infer https://fbinfer.com/, CodeQL https://codeql.github.com/.

[10]CompCert [27], seL4 [25], Project Everest [28], [29]

Example 6. Greenman et al. [33] studied how researchers and learners understand LTL properties and found that the encoding of theorems in English language as LTL formulas "was fraught with errors and provides evidence for a large number of misconceptions". Murray and Oorschot [7] compare the user's interpretation of the formal statement to reading the fineprint in legal contracts. For instance, they report that even a leading researcher in the field, Benjamin Pierce, required a full week to understand the functional correctness proof of SeL4 deeply enough to prepare two lectures about them in a graduate seminar [7].

In summary, it is difficult to draw a concrete line between a secure and an insecure system precisely enough to prevent an attacker from exploiting security flaws despite concrete and credible assurances about the absence of security flaws.

B. Challenges Of Reasoning About Cybersecurity

Assuming we could draw the line between secure and insecure systems precisely enough, we are still fundamentally limited in the way we reason about the behaviors of a given software system with respect to that line. To reason about the behaviors of a software system, we need to *model* its behaviors. From properties of the model, we make claims about properties of the running system. An attacker can attack the system while keeping properties in the model in tact.

The most powerful security approaches reason over some model of the behavior of the software system

- *Provable security* uses the tools of logic, formal methods, and mathematics to derive a formal guarantee about the security of a software system. This includes cryptography, model checking, protocol analysis (e.g., consensus or internet protocols), secure-by-construction [34], proof-carrying code [35], [36], and software verification [37].
- *Static analysis* (semantics-based), like symbolic execution, abstract interpretation, and logic-based static analysis (e.g., Infer, CodeQL, SonarQube, and FindBugs) translates the program's source code (or binary) into a model of computation represented in the available semantics, logic, or theory to check assertions about the behaviors of the system [38].
- *Dynamic analysis*, including fuzzing, compartmentalization, sandboxing, trusted execution environments, model the current execution at some (fixed) level of abstraction [39]–[45].
- *Secure-by-design* refers to techniques to avoid introducing security flaws at the design stage. This includes software engineering efforts, such as ensuring security best practices or *threat modelling* to elicit project-specific security requirements as well as *language-based security* efforts to entirely avoid large classes of security faults, such as memory safety issues, at the OS or programming language level [46], [47].

As we always reason about a *specific* model of the system, an attacker may be able to violate a security property of the real system while keeping properties in the model in tact. For instance, the attacker may violate an assumption of the model, exploit a security flaw at a lower level of abstraction, or exploit flaws that are missed in the model.

1) Assumptions. When reasoning about a software system, we make assumptions about the actual, system as it is running in production. An attacker can violate an assumption. These assumptions are often about matters in the empirical world *outside* the model used to reason about the deployed system.

Example 7. The security of cryptographic protocols depends on *assumptions about the computational resources* of an attacker or the computational complexity of a mathematical problem. For example, the RSA public key encryption is secure assuming that we cannot efficiently factor large numbers into their prime factors. This assumption is broken on a quantum computer with Shor's algorithm. Cryptography, as maturing field with important innovations, has a long history of make-and-break cycles where assumptions are exploited to break protocols otherwise provably secure [5], [48], [49].

Example 8. Assumptions (or axioms) are fundamental in provable security. Bognar et al. [37] analyzed two embedded trusted-execution architectures that were formally shown *provably-secure* [50], [51]. They identified nine assumptions, e.g., (i) Enclave software cannot access unprotected memory or manipulate interrupt functionality. (ii) Interrupted enclaves can only be resumed once with `reti` and not be restarted from ISR. (iii) The `dma_addr` bus contains the full address. (iv) All components use a consistent key size. For every assumption, Bognar et al. demonstrated how it could be exploited to successfully attack the trusted architectures. Kang et al. [52] analyzed a compiler that was formally shown *provably-correct* [27]. They identified an axiom that was empirically invalid and could be exploited such that the compiled binary would compute an unexpected result.

Example 9. In formal reasoning, we can make assumptions about the "model-external" world explicit via *formal contracts*. For instance, *proof-carrying code* [35] makes explicit which properties otherwise untrusted code satisfies for the execution to be secure. Vanegue [36] showed, using weird machines, how these guarantees can be bypassed. *Execution* and *leakage contracts* [53] make assumptions about the hardware explicit (to enforce an operational definition of constant-time).

Example 10. In static analysis, we might make assumptions about the callers of a function if our analysis is *intra*-procedural. We might make assumptions about the maximum number of loop iterations or the maximum depth of recursion. When not all code is available, such as for third-party libraries or system calls, we might choose to model those. We might make assumptions about the maximum size of the inputs or the operational distribution [54] for a software system.

Example 11. Assumptions about components outside the model are also made in *language-based security*. For instance, we assume that calls to `unsafe` in memory-safe languages like Rust or Java, e.g., to interact with the hardware or issue calls to a native C-library, are safe-by-developer. We might also make inconsistent assumptions *across* the language boundary. For instance, Mergendahl et al. [55] demonstrated an attack where control-flow integrity (CFI) is enforced in the C part and memory-safety in the Rust part by corrupting memory in C and hijacking control flow in Rust.

Even if assumptions are stated explicitly, it is sometimes difficult to assess (a) the degree to which an assumption holds for the deployed software system and (b) the resulting threat to the high-level security guarantee.

2) Abstractions. A central idea in computer science, *abstraction* allows us to focus attention on details of greater importance by removing or generalizing physical, spatial, or temporal details or attributes of real-world objects or systems. However, because we must always reason at an (arbitrary but) *specific* level of abstraction, an attacker may be able to violate a security property at a *lower level* of abstraction even if the property was shown to hold at the higher level of our model.

• *Specification vs implementation*. There is a gap between a system's specification and its implementation. Despite demonstrating a security property for the specification, an attacker might still be able to violate it in the implementation.

Example 12. SHA-3 is currently considered as the most secure cryptographic hash function. Yet, a simple buffer overflow in the most widely used implementation of SHA-3 allows an attacker to bypass the security by computing (second) preimages and even to execute arbitrary code on the victim's machine [56].

• *Source code vs compiled binary*. There is a gap between the developer-provided source code and the compiler-provided program binary for that code. Given only the source code, without assumptions about the compiler, it is hard to make reliable statements about properties of the executable.

Example 13. Effectively all memory safety and undefined behavior issues emerge from this abstraction gap. In the presence of behavior where the language standard imposes no requirement (i.e., undefined behavior), the compiler is allowed to do anything it chooses, including "having demons fly out of your nose".[11] In fact, undefined behavior includes memory safety issues, which currently constitute 80% of security vulnerabilities exploited in the wild.[12] Undefined behavior also includes type confusion vulnerabilities where the same unchanged variable is interpreted to have different values in different parts of the program.[13] If the developer decides to handle an instance of undefined behavior, e.g., by checking for an integer overflow in a saturated increment, the compiler might remove the check (because it is itself undefined behavior) but keep the original undefined behavior unhandled.[14]

Example 14. Even if the compiler guarantees *semantic equivalence* between source code and executable, it might be possible to *bypass*, in the executable, security properties established at the source level. D'Silva et al. [57] identified several compiler optimizations, such as dead store elimination, code motion, and function call inlining that are formally guaranteed to be semantic preserving but might (i) introduce information leaks through persistent state, (ii) eliminate security-relevant code, or (iii) introduce side channels. D'Silva et al. explain that "the operational semantics used in correctness proofs includes the state of the program, but not the state of the underlying machine" [57]. When reasoning about a program's behavior, the tool's abstract model may not reflect these detailed compiler choices.[15]

• *Program vs process*. Finally, there is a gap between the program as it is stored in memory and the process as it runs on the machine. Given only the program, without assumptions about the machine, it is hard to make reliable statements about properties of the running process.

Example 15. Effectively all side channels emerges from this gap. Even if we can perfectly guarantee the absence of timing side channels by carefully analyzing the program (binary), an attacker might exploit processor-specific optimizations or microarchitectural features like speculative execution to violate the constant time property [20], [26], [31].

Example 16. All hardware-specific software vulnerabilities and hardware-assisted software security emerges from this gap, as well. For instance, even if we can guarantee information flow security in the program [25], an attacker might exploit defective DRAM memory modules (RowHammer) [7], [18] or a bug in the CPU's microcode [58] to leak sensitive information. The Zenbleed bug allowed anyone who could execute a certain sequence of instructions (e.g., in a sandbox, container, or separate process) on an AMD Zen 2 class processor to read parameters or return values of sensitive functions like strlen, memcpy, and strcmp *anywhere* on that physical machine.

Example 17. It is convenient for the assembly programmer or the compiler engineer to think of the CPU as a blackbox that implements a specific instruction set, like x86. To study this abstraction, Domas [59], [60] implemented a tool to exhaustively search the x86 instruction set that is actually *implemented* and found critical x86 hardware glitches, previously unknown machine instructions, ubiquitous software bugs, and flaws in enterprise hypervisors.

In summary, if we reason about an abstraction of the software system to make claims about the actual, deployed system, an attacker might violate the security property at a lower level of abstraction while keeping the guarantees at the higher level in tact [36], [61], [62]. Or as Balakrishnan and Reps put it: "What you see is not what you execute" [63]. We also note that the specific level of abstraction is often fixed by the language (or logic or theory) that is used in the reasoning framework.

3) Non-Universal claims. An under-approximate analysis only allows existential statements. Approaches like fuzzing, software testing, symbolic execution, runtime verification, or incorrectness logic only allow us to reason about an (observed) subset of all executions. Even if we assume that we know which security properties should hold and how to operationalize them, these approaches do not allow us to make universal claims about the operationalized properties.

[11] https://groups.google.com/g/comp.std.c/c/ycpVKxTZkgw/m/S2hHdTbv4d8J
[12] https://googleprojectzero.blogspot.com/p/0day.html
[13] https://blog.trailofbits.com/2022/11/10/divergent-representations/
[14] https://research.swtch.com/ub
[15] https://news.ycombinator.com/item?id=39191507

Example 18. Despite years of fuzzing the Suricata open-source intrusion detection and prevention system, a one-line division-by-zero bug that could cause denial-of-service was found only by a careful manual audit.[16]

4) Fundamental limits. There are questions about the operational security properties of a software system that we can simply not answer within any sufficiently powerful reasoning framework. This includes approaches in provable security and in semantics-based static analysis.

- From a math perspective, according to Gödel's incompleteness theorem, for any sufficiently complex deductive system that is consistent, there are always true statements that cannot be proved in that system. In other words, a reasoning framework cannot be both, consistent and complete.
- From a verification perspective, according to Rice's theorem all non-trivial semantic properties of programs are *undecidable*. A property is non-trivial if it is neither true for all programs, nor false for all programs.
- From a program analysis perspective, according to Landi [64], [65] there exists no algorithm that—in general and even under simplifying assumptions—could decide whether there exists an execution of the program where two pointers point to the same memory address.
- From a security perspective, according to Cohen, there exists no algorithm that, in general, could detect malware [66].

In practice, theoretical limits have never stopped us from making progress. We allow our analysis to be over- or under-approximate. An over-approximate analysis (e.g., abstract interpretation) reports security flaws that do not exist while an under-approximate analysis (e.g., symbolic execution) misses security flaws that do exist. However, security is a universal claim, and we cannot be sure about the degree to which our heuristic choices impact the analysis result. Worse, we currently do not even know how to *quantify* the loss of guarantee [67]. Even if the analysis is sound, an over-approximate analysis may yield so many false positives that the developer might miss the actual security flaw like a needle in a haystack.

C. Sociotechnical Challenges

1) Infinite regress of trust. We seek reliable statements about a software system because we do not *trust* its security in the first place. Yet, we trust whatever the security of the system depends on (i.e., the software supply chain, the code review process, the build infrastructure, or the hardware, hypervisor, and operating system that runs our system). Even if we do not, the same argument applies recursively. An attacker may attack the software system by exploiting bugs or security flaws in whatever we trust to work as expected.

Example 19. Cryptography ensures the confidentiality of some data (e.g., messages) *only if we trust* the confidentiality of other data (e.g., secret keys). However, secret keys can be compromised. For instance, Intel SGX-based trusted execution environments have recently been compromised because the root provisioning key (SGX Fuse Key0) and the root sealing key (SGX Fuse Key1) can be leaked.[17]

Example 20. We trust the compiler to be non-malicious. In his Turing award lecture [68], Thompson presents a process to create an open-source compiler that can stealthily inject vulnerabilities into the compiled binary: In Stage 1, we write a compiler that can compile itself. In Stage 2, we add code to inject a vulnerability into the compile**d** binary and itself into the compile**r** binary. Then, we recompile the compiler. In Stage 3, we remove the offending code from the open-source compiler, recompile, and distribute the resulting compiler binary. The compiler binary will eternally inject the vulnerability-injecting code when compiling its own vulnerability-free source. This is Thompson's reflection on the trust we put into our tools and the people writing these tools.

Example 21. Project maintainers rely on a code review process to check the absence of bugs in new contributions. While we can safely assume that the vast majority of contributions are benign, without additional tooling there are opportunities for an attacker to exploit the developer's trust in code contributions and inject vulnerabilities that could later be exploited [69].[18]

Example 22. We trust that our automated tooling, including compilers, linkers, debuggers, fuzzers, static analysers, verifiers, and theorem provers, work as advertised. For instance,

- Even if we trust the security policies of the linux kernel, there may be bugs in the *eBPF verifier*[19] preventing us from detecting policy violations [70]–[72].
- Even if we trust the fuzzer, there may be bugs in the *sanitizer* which would prevent the fuzzer from flagging security flaws that are actually in scope [73].
- Even if we trust the software verifier, there may be bugs in the *constraint solver* that is used which could undermine the provided formal guarantee [74], [75].

Problem of Infinite Regress [76]. In epistemic philosophy, this is the problem of Infinite Regress: A belief is justified because it is based on another belief that is justified. We must anchor our trust somewhere. While these trust anchors are often well-specified in security, they are not exempt from scrutiny and subject to the same challenges we have listed in support of our theory. An attacker may be able to attack the system by exploiting bugs or security flaws in our trust anchors.

2) Competing points of view. Security is a primary concern only for us. There are many stakeholders in a software system. These stakeholders have different *perspectives* on the system that might (often unintentionally) not include security. Stakeholders often have different *requirements* for the software system which are sometimes conflicting with its security. An attacker may exploit design decisions made to accommodate non-security preferences of other stakeholders.

[16] https://twitter.com/catenacyber/status/1646860408014118913

[17] https://x.com/PratyushRT/status/1828183761055330373

[18] XZ open source attack (Jia Tan): https://research.swtch.com/xz-timeline

[19] eBPF (https://eBPF.io) enables users to instrument a running system by loading small programs into the operating system kernel. It is used, e.g., to implement security policies.

Example 23. While the developer takes a *constructive perspective*, the attacker takes an *adversarial perspective* [21]. On the one hand, the *developer* reads code to implement *intended* features and to fix bugs. On the other hand, the *attacker* reads code (or binary) to exploit *unintented* features (i.e., the weird machine) [36], [61], [62]. These conflicting perspectives must be reconciled. The developer implements with intention but might realize something that deviates from this intention [77]. This difference between intention and realization can be exploited "to program the weird machine" [62].

Example 24. *Better performance* often wins against better security. For instance, the browser security team of Microsoft Edge recently found that disabling the Just-In-Time (JIT) optimization could mitigate half of the Chrome exploits observed in-the-wild.[20] In fact, performance optimization is a root cause of many critical types of security flaws. For instance, *memory corruption* (such as buffer overflow or use-after-free) is often caused by undefined behavior which is deliberately left undefined in the C language specification for optimization reasons. *Side channel attacks* are often enabled by microarchitectural or compiler optimizations. For some critical types of security flaws, *we even have mitigations*. However, for performance reasons they are often left disabled in production.

Example 25. Competing points of view might result in stakeholder requirements that conflict with security.

- The *vendor* cares about time-to-market where the software system is tested in production and new features are deployed almost instantly.
- The *customer* cares about a low cost. In fact, Woods [78] suggests if buyers cannot distinguish secure from insecure product, then there is incentive to sell insecure products. Software security exists within a *lemons market* [79].
- The *user* cares about performance and a nice user experience, such that the performance overhead of certain security measures is considered impractical.
- The *developer* cares about understanding the source code, such that certain side effects to make code more secure (e.g., more verifiable) are not appreciated [7], [49], [80].

Example 26. Some maintainers remain unconvinced that rewriting components of their project in a memory-safe langauge will help improve the security of the project:

> *"Will I ever rewrite curl in rust? I don't believe in rewrites, no matter which language. I believe in replacing code and fixing components gradually over time. That *could* mean that we have a curl written mostly in rust in 10 years. Or in 20 years. Or not."*
>
> —Daniel Stenberg (curl main developer) [81]

When developing and advocating our software security tooling and processes, we should always consider the different stakeholders and their (potentially competing) perspectives. However, in general competing requirements practically prevent us from providing guarantees about the software system.

[20]https://microsoftedge.github.io/edgevr/posts/Super-Duper-Secure-Mode/

3) Security is a function of incentive. Security is a function of incentive. Without incentive to attack, a system may only appear to be secure, until there is incentive. The incentive is higher the more widely used or the more critical the software system, component, or library is. The incentive can also be artificially generated via red teaming, bug bounty programs, or pwning competitions. Irrespective of the tools, techniques, and processes we have in place to maximize the security of our system—if there is no real (or artificial) incentive to develop exploits and to report these bugs, we will never know about the true insecurity of the system—irrespective of our confidence in its security—*until* there is incentive.

Example 27. The more incentive there is for ethical hackers to find and report security flaws, the more independent scrutiny this system will have undergone.[21] Apple has become one of the most innovative companies in terms of mitigations (KIP, SCIP, PAC, PPL). For several years, despite the hard work and the awesome innovations of the security team at Apple, it took no more than a few weeks from the release of the new iPhone to the next jailbreak. The large market for jailbreaks has provided a strong incentive for hackers. However, when two iOS researchers presented the technical details of the most recent jailbreak at NullCon Goa 2022 [82], they responded to a question on how to get started learning to jailbreak: "It is already too late": While a kernel read/write had been sufficient a few years prior, it was only the starting point eight months before their successful jailbreak that year.

Mitigations as cost. Any mechanism or process to improve the security of a system also increases the cost which subtracts from the incentive. Due to this incentive structure, an attacker will always target low hanging fruits first. Once a mitigation is adopted to prevent a vulnerability class *currently* popular, attackers will just move on to the next low hanging fruit [83].

Security-by-obscurity not viable. According to Kerckhoff's principle, a system should be secure *even **if*** everything about the system is public knowledge. Obfuscation and other deterrents increase the imbalance between attacker and defender in favor of the attacker. While there is obviously no incentive for ethical hackers to discover and responsibly disclose existing security flaws, a determined attacker can be expected to have more skills, resources, and incentive to exploit the (unhandled) security flaws despite the deterrent. The security of a system should not depend on the secrecy of its components. The reputational and financial damage arising from unknown exploits is significantly worse than inviting responsible disclosure via an open security approach.

Example 28. Dellago et al. [45] studied the market for 0-day exploits from 2016 and 2021.[22] For the four studied operating systems (Windows, MacOS, iOS, and Android), the number of 0-days observed in the wild decreased from 2016 to 2018 but increased again until the end of the study period. For the two investigated exploit brokers, Zerodium and Crowdfence, exploit prices increased to a maximum in 2019 and have since

[21]This also means that the number of security flaws that are *known* for this system might be a measure more of its criticality than its insecurity.

[22]0-day exploits use vulnerabilities that are unknown to the developers.

remained stable at USD 2.5-3 *million* for a full exploit chain with persistence on Android or iOS. Recently, a third broker, Operation Zero, has announced to offer USD 20 *million* for the same, which Google's Senior Director of the Threat Analysis Group, Shane Huntley, took as a good sign that the developed mitigations are becoming more effective and exploits more difficult.[23] Schechter [84] confirms that higher prices are a signal of more secure products.

Economic and legal guarantees. Even if a software system is formally certified to be secure and no security flaw is *known*, it might still "become" insecure once there is enough incentive to develop an attack. A lack of incentive or visibility practically prevents us from thoroughly assessing the security of a system. An economic or legal framework can provide such incentives. For instance, a *bug bounty program* guarantees (towards all stakeholders) that the software system is at least as secure as they would pay someone else for reporting a security flaw in the system [78], [84]. A bug bounty program also serves as a signal to the vendor about the security of the system and effectively improves security, too.

IV. A Philosophy of Security

While our *theory of guarantees* is meant to *explain* why we must generally assume that exploitable security flaws exist in any sufficiently complex and widely-used software system, our *philosophy of security* is meant to *serve as a guiding principle* in the acknowledgement of this conclusion.

Consequences. What does our theory mean for cybersecurity as a research field? Firstly, our theory systematically elucidates the fundamental attack surface of *any* software system which which, hence identified, can now be hardened in a concrete and directed manner. Secondly, our theory explains why research in cybersecurity might seem reactive, specific, and ephemeral: Defenses are always built as a reaction to and specific to one type of attack, and vice versa. Lastly, our theory motivates a new philosophy of security.

Our philosophy. How to maxmize the reliability of our statements about the world in the absence of a reliable approach? In his philosophy of science, Popper suggests that a scientific theory must evolve in a counterexample-guided manner via *falsification* [3]. In his philosophy of maths, Lakatos suggests that even mathematical theorems evolve in a counterexample-guided manner via *proofs and refutations* [4]. Similarly, we suggest in our philosophy of security that claims of security must evolve in a counterexample-guided manner, too.

> *"For Lakatos, the development of mathematics should not be construed as series of deductions [..]. Rather, these water-tight deductions from well-defined premises are the (perhaps temporary)* ***end-points*** *of an evolutionary, and indeed a* ***dialectical****, process in which the constituent concepts are initially ill-defined, open-ended or ambiguous but become sharper and more precise in the context of a protracted debate."*
>
> —Musgrave and Pigden (Stanford Encyclopedia of Philosophy) [17]
> (emphasis ours)

[23] https://twitter.com/ShaneHuntley/status/1706944206521160070

Problem of Induction [85]. The prevalent philosophy of security has been to develop security approaches with *inductive* claims about their effectiveness, i.e., to demonstrate the *generality* of the security approach: When a new security technique or process T is developed, we deem it as effective if no security flaw is missed in an evaluation. However, such inductive, universal claims can never be confirmed despite all evidence in favor [5]. Just *one* identifed security flaw becomes undisputable evidence against any claim of security that arises from T [61]. In fact, our theory suggests that, fundamentally, *no* such approach can exist. Hence, our philosophy suggests a different method with a focus is on identifying and remediating the *specific limits* of a technique (which implements the security approach) instead of its generality. We can never claim that a technique is effective in absolute terms.

A. Vulnerability-Guided Hardening of Security Approaches

We propose a counterexample-guided improvement of our security techniques and processes to gradually maximize their effectiveness. We should always seek to find evidence *against* the generality of the technique, and reconcile such counterexamples with that technique. Specifically, we suggest

1) Implement a security approach T to protect software S.
2) Find evidence ce *against* the security of S despite T (e.g., from bug reports solicited via a bug bounty program).
3) Patch S and improve T by (i) debugging what made T ineffective in the presence of ce, and (ii) changing T to make it effective for the most general version of ce.
4) Go to (2).

But how does this method address the identified challenges?

Operationalization (§III-A). Our philosophy addresses the first challenge of finding which properties must hold in the first place (unknown unknowns) and of drawing the right line with the right pen by soliciting (and reacting to) independent vulnerability reports, e.g., via bug bounty programs, red teaming, auditing, or pwning competitions [1]. If a vulnerability exploits a behavior we have not previously considered as a violation of a security property or if it exploits an improper operationalization, we improve our techniques to detect, prevent, or mitigate similar types of vulnerabilities. In this sense, our method offers an adversarial perspective [21] on our security tooling and processes (in addition to the system itself).

Reasoning (§III-B). Our philosophy addresses the challenge of reasoning about these operational properties at the right level of abstraction using empirically valid assumptions and a sufficiently expressive reasoning framework. Vulnerability discovery (Step 2) can be guided (i) by systematically uncovering implicit and violating explicit assumptions and (ii) by systematically exploring the gap between the software system and the abstractions used by our tools. If a discovered vulnerability exploits one of those those, (Step 3) we improve the technique by adjusting the assumptions and abstractions, or by improving the expressibility of the language, logic, or theory. In this sense, our method offers systematic guidance for improving our security tooling and processes.

Algorithm 1 Counterexample-guided Meta-Verification

Input: Initial software system S
Input: Initial formalized assumptions A and properties P
Input: Initial formal framework F
1: $\langle S, A, P, F\rangle = \texttt{verify}(S, A, P, F)$
2: Counterex. $ce = \texttt{attack}(S, A, P, F)$
3: **while** ce exists **do**
4: $\quad\langle S, A, P, F\rangle = \texttt{reconcile}(ce, S, A, P, F)$
5: $\quad\langle S, A, P, F\rangle = \texttt{verify}(S, A, P, F)$
6: $\quad ce = \texttt{attack}(S, A, P, F)$
7: **end while**
Output: Verified software system S
Output: Updated assumptions A and properties P
Output: Empirically more valid formal framework F

Sociotechnics (III-C). Our philosophy, and specifically soliciting independent vulnerability reports, addresses the remaining challenge of establishing trustworthiness and reconciling competing points of view. For instance, an effective bug bounty program exploits security as a function of incentive, does not require any assumption of trust, and signals to all stakeholders how much money can be offered for a vulnerability without getting any actual vulnerabilities reported. In fact, we could consider a bug bounty program as an *economic guarantee of security* [45], [78], [84]. Once a vulnerability is reported, it can be used to harden our in-house security techniques and processes.

B. Example: Counterexample-guided Meta Verification

To demonstrate the application of our vulnerability-guided hardening method, we instantiate it concretely for software verification which is traditionally presented as *concluding with a proof of security*. Instead of concluding with a proof, we propose to embed the proof process explicitly into a prove-and-break feedback loop. Successful attacks on the actual software system serve as counter-examples that must be systematically reconciled into the formalization. As this counterexample-guided approach allows us to verify the verification process itself, we call this approach as "adversarial" verification or counterexample-guided *meta verification*.[24]

If a critical system can be attacked despite a formal proof of its security, what then is the utility of provable security? Formally verified software like SeL4 [25] or CompCert [27] are among the most secure, high-assurance systems to ever exist. While the process of proof already rids the program of many bugs [87], the final proof inherently excludes large classes of security flaws and provides a precise formal guarantee. The provided guarantee holds within an explicit formalization and with respect to precisely stated assumptions and security requirements [5]. Indeed, it is those artifacts whose empirical validity we propose to systematically scrutinize and harden.

[24]A related procedure is counterexample-guided abstraction refinement (CEGAR) [86]. In meta verification, the counterexample violates the property at the highest level of *refinement* (i.e., the deployed system) but not in the abstraction. In CEGAR, it is the other way around: the counterexample violates the property in the *abstraction* but may not in a refinement.

Algorithm 1 shows our proposed counterexample-guided meta verification procedure. We invite the research community to develop systematic approaches for the individual functions. The procedure starts with the verification of the software system S in Line 1, using an initial set of assumptions A and properties P formalized within the logic or theory of the verification framework F.

However, in practice and in our proposed procedure, the verification (i.e., the function `verify`) does *not end with the failure to verify*, but continues with systematic adjustments to S, A, P, or F until the verification finally succeeds (returning an updated set $\langle S, A, P, F\rangle$). Once S is successfully verified w.r.t. A and P within F, we suggest to `attack` S to identify a security flaw ce that still exists in the actual system S *despite* the successful verification of its formal model (Line 2). As long as we can find successful counterexamples ce, we must update the system and verification artifacts to `reconcile` the counterexample with the formal model of the system and re-verify using the updated verification artifacts (Lines 3–7).

It is interesting to note that meta verification effectively tackles the well-known argument of De Millo, Lipton, and Perlis [6] against the utility of automated verification. The authors argue that the validity of a formal proof is usually derived within a *social process*, and suggest that automated verification might remove this social process and thus the means by which to establish a proof's validity. Meta verification addresses their argument by substituting the social process with a programmatic falsification of the verification result until no more counterexample ce can be found.

1) Verification-guided Vulnerability Discovery. Once the software system has been fully verified, we suggest to try and attack the software system despite the formal proof of security. Apart from existing vulnerability discovery techniques (§3.2), we propose to systematically and perhaps automatically exploit the challenges of formal reasoning about software security so as to "harden" the formal guarantee, the verification artifacts, and the software system.

To implement `attack`, we might identify black swan properties $p \notin P$ or exploit an infidelity in the operationalization of some $p \in P$. For any assumption $a \in A$, we might seek to invalidate a for the actual system [88], [89]. Given the formal framework F, we might seek to violate a property $p \in P$ for S while keeping p in the abstract model of S in tact. To increase public *trust* in tooling and framework F, we propose to invite all stakeholders to subject F itself to bug finding, and to maximize the layman's comprehension of the provided formal guarantees (or "fine print" [7]).

2) Counterexample-guided Reformalization. Once a violation ce of a property $p \in P$ is identified in the deployed software system S *despite* the formal guarantees provided by the formal reasoning framework F modulo the assumptions A, we must update the verification artifacts accordingly. To implement `reconcile`, we suggest to identify the verification artifacts $\langle A, P, F\rangle$ that are inconsistent with the counterexample and systematically and perhaps automatically update

the artifact(s) to enforce a property violation in the presence of ce for the unchanged S. For instance, we might add new properties to P, remove invalid assumptions from A, or update the reasoning framework F, e.g., by fixing a bug in the tooling [74] or by extending the reasoning rules (i.e., logic [31] or assurance cases [90]). Once the formalization recognizes ce, the software system S should be fixed, re-verified, and attacked until no counterexample can be found (Lines 3–7; Alg. 1).

ACKNOWLEDGMENTS

This paper was written out of a deep curiousity to understand the reliability of the statements that our approaches can generate about the properties of a software system, and how we should proceed if we accepted that these statements are fundamentally unreliable from a formal point of view. On the one hand, researchers perpetually promise guarantees and effectiveness. On the other hand, practitioners are troubled by the obvious lack of guarantees. This paper is an attempt to reconcile the promise with reality: *Even a formal security proof must end with the invitation to find new vulnerabilities*.

We extend our special thanks to Toby Murray, Joshua J. Drake, Daniel Woods, Nathanial (d0nut) Lattimer, and Thomas (Halvar Flake) Dullien for the insightful feedback and inspiring discussions.

REFERENCES

[1] Z. D. Initiative, "Results of the pwn2own hacking contest," 2024, https://www.zerodayinitiative.com/blog/2024/3/20/pwn2own-vancouver-2024-day-one-results https://www.zerodayinitiative.com/blog/2024/3/21/pwn2own-vancouver-2024-day-two-results.

[2] C. A. R. Hoare, "Programs are predicates," in *Proc. of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*. USA: Prentice-Hall, Inc., 1985, p. 141–155.

[3] K. R. Popper, *The Logic of Scientific Discovery*. London: Routledge, 1959.

[4] I. Lakatos, *Proofs and refutations*. Nelson London, 1963.

[5] C. Herley and P. Van Oorschot, "Sok: Science, security and the elusive goal of security as a scientific pursuit," in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. S&P 2017, 2017, pp. 99–120.

[6] R. A. De Millo, R. J. Lipton, and A. J. Perlis, "Social processes and proofs of theorems and programs," *Communications of the ACM*, vol. 22, no. 5, p. 271–280, may 1979.

[7] T. Murray and P. van Oorschot, "Bp: Formal proofs, the fine print and side effects," in *Proceedings of the IEEE Cybersecurity Development*, ser. SecDev 2018, 2018, pp. 1–10.

[8] D. Geer, "For good measure: Paradigm," *USENIX login*, vol. 41, no. 3, 2016.

[9] B. Blakley, "The emperor's old armor," in *Proceedings of the 1996 Workshop on New Security Paradigms*, ser. NSPW '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 2–16.

[10] D. Evans and S. Stolfo, "Guest editors' introduction: The science of security," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 16–17, 2011.

[11] S. Peisert and M. Bishop, "I am a scientist, not a philosopher!" *IEEE Security & Privacy*, vol. 5, no. 4, pp. 48–51, 2007.

[12] S. Xu, M. Yung, and J. Wang, "Seeking foundations for the science of cyber security," *Information Systems Frontiers*, vol. 23, pp. 1–5, 2021.

[13] F. B. Schneider, "Blueprint for a science of cybersecurity," *The Next Wave*, vol. 19, no. 2, p. 47–57, 2012.

[14] N. Koblitz and A. Menezes, "Another look at provable security," https://www.math.uwaterloo.ca/~ajmeneze/anotherlook/, acc: 2023-03-09.

[15] ——, "Another look at provable security ii," in *Proceedings of the 7th International Conference on Cryptology in India*, ser. INDOCRYPT'06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 148–175.

[16] ——, "Critical perspectives on provable security: Fifteen years of "another look" papers," Cryptology ePrint Archive, 2019. [Online]. Available: https://eprint.iacr.org/2019/1336

[17] A. Musgrave and C. Pigden, "Imre Lakatos," in *The Stanford Encyclopedia of Philosophy*, Spring 2023 ed., E. N. Zalta and U. Nodelman, Eds. Metaphysics Research Lab, Stanford University, 2023.

[18] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *ISCA'14*, 2014, p. 361–372.

[19] T. Hunter and G. de Vynck, "The 'most serious' security breach ever is unfolding right now. here's what you need to know," *Washington Post*, December 2021.

[20] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.

[21] N. Young and S. Krishnamurthi, "Early post-secondary student performance of adversarial thinking," in *ICER 2021*, New York, NY, USA, 2021, p. 213–224.

[22] L. Ren, "Analysis of nakamoto consensus," Cryptology ePrint Archive, Paper 2019/943, 2019. [Online]. Available: https://eprint.iacr.org/2019/943

[23] J. Neu, E. N. Tas, and D. Tse, "The availability-accountability dilemma and its resolution via accountability gadgets," in *Financial Cryptography*, ser. Lecture Notes in Computer Science, vol. 13411. Springer, 2022, pp. 541–559.

[24] ——, "Ebb-and-flow protocols: A resolution of the availability-finality dilemma," in *IEEE Symposium on Security and Privacy*. IEEE, 2021, pp. 446–465.

[25] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "Sel4: Formal verification of an os kernel," ser. SOSP '09, New York, NY, USA, 2009, p. 207–220.

[26] G. Barthe, M. Böhme, S. Cauligi, C. Chuengsatiansup, D. Genkin, M. Guarnieri, D. Romero, P. Schwabe, D. Wu, and Y. Yarom, "Testing side-channel security of cryptographic implementations against future microarchitectures," in *Proceedings of the 31st ACM Conference on Computer and Communications Security*, ser. CCS'24, 2024.

[27] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, p. 107–115, jul 2009. [Online]. Available: https://doi.org/10.1145/1538788.1538814

[28] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M. Wintersteiger, and S. Zanella-Béguelin, "Evercrypt: A fast, verified, cross-platform cryptographic provider," in *IEEE Symposium on Security and Privacy*. IEEE, May 2020.

[29] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *CCS'17*, New York, NY, USA, 2017, p. 1789–1806.

[30] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "Sok: Practical foundations for software spectre defenses," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, May 2022.

[31] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *IEEE Symposium on Security and Privacy*. IEEE, 2021, pp. 1868–1883.

[32] N. Mehrabi, F. Morstatter, N. Saxena, K. Lerman, and A. Galstyan, "A survey on bias and fairness in machine learning," *ACM Comput. Surv.*, vol. 54, no. 6, jul 2021.

[33] B. Greenman, S. Saarinen, T. Nelson, and S. Krishnamurthi, "Little tricky logic: Misconceptions in the understanding of ltl," *The Art, Science, and Engineering of Programming*, vol. 7, no. 7, pp. 1–37, 2023.

[34] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ser. ICSE '10, New York, NY, USA, 2010, p. 215–224.

[35] G. C. Necula, "Proof-carrying code," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '97, New York, NY, USA, 1997, p. 106–119.

[36] J. Vanegue, "The weird machines in proof-carrying code," in *Proceedings of the 2014 IEEE Security and Privacy Workshops*, 2014, pp. 209–213.

[37] M. Bognar, J. Van Bulck, and F. Piessens, "Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1638–1655.

[38] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static C code analyzers for vulnerability detection," in *ISSTA*. ACM, 2022, pp. 544–555.

[39] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications confining the wily hacker," in *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, ser. SSYM'96. USA: USENIX Association, 1996, p. 1.

[40] M. Brand, "Virtually unlimited memory: Escaping the chrome sandbox," https://googleprojectzero.blogspot.com/2019/04/virtually-unlimited-memory-escaping.html, Apr. 2019, accessed: 2023-09-01.

[41] V. Costan and S. Devadas, "Intel sgx explained," Cryptology ePrint Archive, Paper 2016/086, 2016. [Online]. Available: https://eprint.iacr.org/2016/086

[42] H. Lefeuvre, V. Badoiu, Y. Chen, F. Huici, N. Dautenhahn, and P. Olivier, "Assessing the impact of interface vulnerabilities in compartmentalized software," in *NDSS*. The Internet Society, 2023.

[43] S. Fei, Z. Yan, W. Ding, and H. Xie, "Security vulnerabilities of sgx and countermeasures: A survey," *ACM Comput. Surv.*, vol. 54, no. 6, jul 2021.

[44] A. Nilsson, P. N. Bideh, and J. Brorsson, "A survey of published attacks on intel SGX," *CoRR*, vol. abs/2006.13598, 2020.

[45] M. Dellago, D. Woods, and A. Simpson, "Characterising 0-day exploit brokers," in *Proceedings of the 21st Workshop on the Economics of Information Security*, ser. WEIS 2022, Jun. 2022.

[46] P. T. Devanbu and S. Stubblebine, "Software engineering for security: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00, 2000, p. 227–239.

[47] L. Kohnfelder and P. Garg, "The threats to our products," *Microsoft Interface*, April 1999.

[48] N. Koblitz and A. Menezes, "Critical perspectives on provable security," *Advances in Mathematics of Communications*, vol. 13, no. 4, pp. 517–558, 2019. [Online]. Available: /article/id/5d2d9acd-8f7a-4e46-8e4a-dfab9701f215

[49] D. E. Denning, "The limits of formal security models," https://faculty.nps.edu/dedennin/publications/National%20Computer%20Systems%20Security%20Award%20Speech.htm, 1999, accessed: 2023-03-09.

[50] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling, "Sancus 2.0: A low-cost security architecture for iot devices," *ACM Trans. Priv. Secur.*, vol. 20, no. 3, jul 2017.

[51] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "Vrased: A verified hardware/software co-design for remote attestation," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 1429–1446.

[52] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis, "Lightweight verification of separate compilation," *SIGPLAN Not.*, vol. 51, no. 1, p. 178–190, jan 2016.

[53] Z. Wang, G. Mohr, K. von Gleissenthall, J. Reineke, and M. Guarnieri, "Specification and verification of side-channel security for open-source processors via leakage contracts," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23, 2023, p. 2128–2142.

[54] J. Geldenhuys, M. B. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. Association for Computing Machinery, 2012, p. 166–176.

[55] S. Mergendahl, N. Burow, and H. Okhravi, "Cross-language attacks," in *NDSS*. The Internet Society, 2022.

[56] N. Mouha and C. Celi, "A vulnerability in implementations of sha-3, shake, eddsa, and other nist-approved algorithm," Cryptology ePrint Archive, Paper 2023/331, 2023. [Online]. Available: https://eprint.iacr.org/2023/331

[57] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *Proceedings of the IEEE Security and Privacy Workshops*, ser. LangSec 2015, 2015, pp. 73–87.

[58] T. Ormandy, "Zenbleed," https://lock.cmpxchg8b.com/zenbleed.html, accessed: 2021-03-12.

[59] C. Domas, "Breaking the x86 isa," 2017, https://www.blackhat.com/docs/us-17/thursday/us-17-Domas-Breaking-The-x86-Instruction-Set-wp.pdf.

[60] ——, "God mode unlocked: Hardware backdoors in redacted x86," 2018, https://www.youtube.com/watch?v=jmTwlEh8L7g&ab_channel=DEFCONConference.

[61] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit programming: From buffer overflows to weird machines and theory of computation," *USENIX ;login:*, vol. 36, no. 6, Dec 2011.

[62] T. Dullien, "Weird machines, exploitability, and provable unexploitability," *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 2, pp. 391–403, 2020.

[63] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, aug 2010. [Online]. Available: https://doi.org/10.1145/1749608.1749612

[64] W. Landi, "Undecidability of static analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, p. 323–337, dec 1992.

[65] G. Ramalingam, "The undecidability of aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, p. 1467–1471, sep 1994.

[66] F. Cohen, "Computer viruses: Theory and experiments," *Computers & Security*, vol. 6, no. 1, pp. 22–35, 1987.

[67] M. Böhme, "Statistical reasoning about programs," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE 2022, 2022.

[68] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, p. 761–763, aug 1984.

[69] K. Lu and Q. Wu, "On the feasibility of stealthily introducing vulnerabilities in open-source software via hypocrite commits," 2021.

[70] S. Bhat and H. Shacham, "Formal verification of the linux kernel ebpf verifier range analysis," Technical Report, 2022. [Online]. Available: https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf

[71] S. Scannell, V. Palmiotti, and J. J. L. Jaimez, "Alice in kernel land: Lessons learned from the ebpf rabbit hole," BlackHat Asia'23, 2023. [Online]. Available: https://i.blackhat.com/Asia-23/AS-23-Palmiotti-Alice-In-Kernel-Land-wp.pdf

[72] J. Jia, R. Sahu, A. Oswald, D. Williams, M. V. Le, and T. Xu, "Kernel extension verification is untenable," in *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, ser. HOTOS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 150–157. [Online]. Available: https://doi.org/10.1145/3593856.3595892

[73] S. Li and Z. Su, "Ubfuzz: Finding bugs in sanitizer implementations," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '24, 2024, p. 14.

[74] M. N. Mansur, M. Christakis, V. Wüstholz, and F. Zhang, "Detecting critical bugs in smt solvers using blackbox mutational fuzzing," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 701–712. [Online]. Available: https://doi.org/10.1145/3368089.3409763

[75] D. Winterer, C. Zhang, and Z. Su, "Validating smt solvers via semantic fusion," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 718–730. [Online]. Available: https://doi.org/10.1145/3385412.3385985

[76] R. Cameron, "Infinite Regress Arguments," in *The Stanford Encyclopedia of Philosophy*, Fall 2022 ed., E. N. Zalta and U. Nodelman, Eds. Metaphysics Research Lab, Stanford University, 2022.

[77] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 57–72. [Online]. Available: https://doi.org/10.1145/502034.502041

[78] D. W. Woods, "Lemons and liability: Cyber warranties as an experiment in software regulation," in *Blackhat*, 2023.

[79] G. A. Akerlof, "The market for "lemons": Quality uncertainty and the market mechanism," *The Quarterly Journal of Economics*, vol. 84, no. 3, pp. 488–500, 1970.

[80] D. Rosenblum, "A practical approach to programming with assertions," *IEEE Transactions on Software Engineering*, vol. 21, no. 1, pp. 19–31, 1995.
[81] D. Stenberg, "Tweet," https://twitter.com/bagder/status/1360131939794042884, accessed: 2023-09-01.
[82] CoolStar and Tihmstar, "Jailbreaking ios in the postapocalyptic era," NullCon Goa, 2022.
[83] H. Marco-Gisbert and I. Ripoll-Ripoll, "Exploiting linux and pax aslr's weaknesses on 32- and 64-bit systems," in *Blackhat Asia*, 2016.
[84] S. Schechter, "How to buy better testing using competition to get the most security and robustness for your dollar," in *Infrastructure Security*, G. Davida, Y. Frankel, and O. Rees, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 73–87.
[85] L. Henderson, "The Problem of Induction," in *The Stanford Encyclopedia of Philosophy*, Winter 2022 ed., E. N. Zalta and U. Nodelman, Eds. Metaphysics Research Lab, Stanford University, 2022.
[86] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, E. A. Emerson and A. P. Sistla, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169.
[87] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532
[88] H. Armstrong and M. Bishop, "Uncovering assumptions in information security," in *Proceedings of the WISE4 Forth World Conference on Information Security Education*, 2005.
[89] J. P. Degabriele, K. Paterson, and G. Watson, "Provable security in the real world," *Proceedings of the IEEE Security and Privacy*, vol. 9, no. 3, pp. 33–41, 2011.
[90] M. Chechik, "On safety, assurance, and reliability: a software engineering perspective (keynote)," in *ESEC/SIGSOFT FSE*. ACM, 2022, p. 2.