Formal Trust and Threat Modeling Using Large Language Models

Zhihao Yao
Computer Science Department
New Jersey Institute of Technology
Newark, New Jersey, USA
zhihao.yao@njit.edu

***Abstract*—Security modeling, including trust and threat modeling, is a critical process of modern system design and analysis. However, the models are often described in imprecise natural languages, and their inconsistent interpretations and implementations can lead to cybersecurity incidents. In this work, we first introduce an extended Linear Temporal Logic to model the multi-faceted security model of a system to capture its temporal and spatial properties and security guarantees. Then, we manually write 10 security model formulas of real-world systems and attack scenarios. Finally, we fine-tune a large language model with our manually written models. We evaluate the fine-tuned model with another set of 9 recent system designs to validate its capability in accurately capturing their security models. Our work provides a formal approach to system security modeling, and it demonstrates the benefits of using large language models in capturing the models of real-world systems.**

***Index Terms*—Large Language Models, Trust Modeling, Threat Modeling, Formal Methods**

1. Introduction

Security modeling is a critical task that requires in-depth understanding of a system's architecture, components, and their interactions with both intra and inter-system entities. Modern computer systems, such as wearables, Internet of Things (IoT) devices, smartphones and tablets, interact with a plethora of entities, including other mobile devices, edge services, cloud services, and human users. Unfortunately, the trust and threat of system security research are largely described in imprecise natural language, leading to inconsistent interpretations and implementations.

Modeling a system's security is a challenging task due to the complexity and diversity of modern systems. For example, the code quality of device drivers is worse than the rest of the operating system kernel: bugs in device drivers constitute 85% of the total Linux kernel bugs [1], [2]. In 2019, a study discovered vulnerabilities in more than 40 device drivers from over 20 different vendors, exposing computers to persistent malware, even though these device drivers are all certified by valid Certificate Authorities [3]. The trust assumptions on device drivers are challenged by the proliferation of I/O devices and their drivers' complexity, and the trust on sandboxing mechanisms are challenged by the increasingly complex inter-privilege-level interactions [4]–[7].

In addition, modern system hardware is equipped with a variety of hardware permission levels, such as trusted execution environments (TEEs) [8]–[12] and physical isolation mechanisms [13]–[17]. The dynamic inter-privilege-level interactions (through syscall, hypercall, security monitor call, mailbox message, etc.) make dataflow-based modeling extremely challenging.

In this work, we propose a novel approach to formally capture the trust and threat models of a system using a fine-tuned large language model (LLM) that is trained on a set of manually created set of security models of real-world systems. We describe the properties of adversaries in our extended Linear Temporal Logic (LTL) formulas, to reflect the spatial and temporal properties of an adversary's capabilities. LTL formulas are often used to verify the correctness of specifications [18]. We use LTL formulas to provide a formal definition of the adversaries, and open the possibility of using model checking tools to further verify the security guarantees of system designs. Corresponding to the threat modeling we define the necessary trust in system security guarantees. These guarantees are formulated as security objectives with goals to protect the confidentiality, integrity, and availability of a system component. A security model combines the trust and threat models to provide a comprehensive view of the system's trust assumptions.

The key contributions of this work are as follows:

- We introduce an extended LTL logic to model the multi-faceted trust and threat models of a system.
- We manually write a set of ten security models in our extended LTL formulas for real-world systems and attack scenarios.
- We fine-tune an LLM with our manually-written formulas.
- We evaluate the fine-tuned LLM with a set of recent works to show the model's capability in accurately capturing their security models.

 Digital Object Identifier (DOI): 10.1109/ACSACW65225.2024.00033

2. Background

2.1. Trust and Threat Modeling

A key challenge in security modeling is the lack of formal definitions and tools to represent and analyze a system. On the one hand, security researchers come up with ad-hoc and highly theoretical assumptions that are difficult to apply in practice [19]. On the other hand, security practitioners often rely on informal, high-level descriptions that are not concrete enough to be useful for security analysis [20]. Recent data breaches and security incidents [21]–[24] have highlighted the importance of improving system security modeling to ensure that the systems are secure and resilient by design. Dataflow-based modeling [25] is a promising approach to address these challenges, as it tracks the origin and propagation of data in a system, creating concrete paths for trust and threat analysis. However, the inconsistency in code quality and the complexity of system software make it difficult to manually create and reason about the dataflow.

2.2. Linear Temporal Logic (LTL)

As adversaries in security assumptions are usually not instantaneous, temporal logic is necessary to model the adversary's capabilities over time. LTL [18], [26] is a standard formalism for specifying such properties. Our LTL annotations are extended on top of these basic operators: $\bigcirc\varphi$ denotes φ is True in the immediate next state; $\Box\varphi$ denotes φ is always True; $\Diamond\varphi$ denotes φ is eventually True; $\psi \boldsymbol{U} \varphi$ denotes ψ remains True at least until φ is True; $\psi \boldsymbol{R} \varphi$ denotes φ remains True until φ first becomes True. ψ and φ can be any state.

3. Formal Threat Modeling

In this section, we define the threat model in LTL formulas to reflect the properties of a potential adversary.

3.1. Adversary Spatial Privilege Levels

The capabilities of an adversary describe to what degree it can perform certain operations, such as executing a program at a specific privilege level or physically manipulating the hardware, and the temporal presence of the adversary, such as persistent after a reboot. Subscripts denote the adversary's spatial privilege level, and superscripts denote the temporal presence of the adversary. The listing below is not meant to be exhaustive, but to provide a general framework and examples for modeling adversaries.

1) $Adv_{Sandbox}$: The adversary is confined by a sandbox, with limited access to a subset of operating system functionalities, which are either implicitly granted by the operating system, or explicitly granted by user.
2) Adv_{User}: The adversary operates as a regular, unsandboxed, user-mode application confined by user account permissions, without the elevated privileges of root access.
3) $Adv_{UserRoot}$: The adversary operates with root privileges in the user mode. Depending on the operating system, the adversary may have access to the entire file system and the permission to adjust system settings.
4) Adv_{Kernel}: The adversary can run code in the kernel mode. For example, the adversary may be a trojan compiled into the kernel, an installable kernel module, or a user-mode application that exploits kernel vulnerability for privilege escalation.
5) $Adv_{Hypervisor}$: The adversary executes code in the hypervisor mode. It may be a trojan in the hypervisor, or a non-hypervisor application that exploit vulnerability to escalate its privilege to the hypervisor mode.
6) Adv_{TEE}: The adversary is an enclave application that operates in a hardware-based TEE, such as ARM TrustZone [10] and Intel SGX [12]. Note that some systems may have nested privilege levels within the TEE mode [9], requiring further refinement of the model.
7) Adv_{RoT}: The adversary compromises the platform Root-of-Trust (RoT) to bypass secure boot and impersonate a protected application in remote attestation.
8) $Adv_{Physical}$: The adversary has physical access to a device and can manipulate the hardware, but it cannot access the internal of a CPU or System-on-a-Chip (SoC) without rendering the device inoperable.
9) $Adv_{Substrate}$: The adversary, for example, a hardware trojan, has access to the substrate of a CPU or SoC, such as the cache and in-package scratchpad memory.
10) $Adv_{Proximity}$: The adversary is external and in proximity to a victim system.

Although an adversary at a lower level (i.e., closer to the hardware) usually carries the capabilities at higher levels, the adversary's capabilities are not necessarily linearly hierarchical. For example, an Intel SGX enclave application has full control over the untrusted part of the same application, but it does not have capability to manipulate other applications or the operating system.

3.2. Adversary Temporal Presence Levels

1) $Adv^{Transient}$: The adversary is transient and does not present after a complete power cycle[1].
2) $Adv^{Residue}$: The adversary leaves non-executable residue after a complete power cycle. Even with a cold boot, it is uncommon for bootloaders to zero out all the memory inside and outside a CPU or SoC [13].
3) $Adv^{Persist}$: The adversary persists after a complete power cycle. It gains persistence by installing itself in the boot process.
4) Adv^{*}: The adversary exists at any temporal presence level.

We model an adversary, which is either external to the system (e.g., a malicious app or a remote attacker exploiting

1. A complete power cycle must include a cold boot, which fully executes all bootloaders from a powered-off state.

a vulnerability) or internal to the system (e.g., a software or hardware trojan), to have a combination of spatial and temporal privilege levels. For example, an $Adv_{Kernel}^{Transient}$ adversary can execute code in kernel mode but does not persist after a reboot; an $Adv_{Hypervisor}^{Residue}$ adversary can execute code in hypervisor mode and leaves non-executable residue after a reboot.

3.3. Adversary's Temporal Progression

An adversary's capabilities is dynamic during an attack. A full system exploit may involve multiple different capabilities over time. At each step, $\mathbf{E}$ denotes an *Exercise* function that maps the adversary's capabilities to the adversary's actions, through exploiting a vulnerability, writing to a filesystem, or other means.

For example, in a recent spyware attack [27], a malicious web app, $Adv_{Sandbox}^{Transient}$, exploits browser vulnerabilities to escape the browser sandbox, becoming $Adv_{User}^{Transient}$, and then, it exploits a kernel vulnerability to escalate its privilege to $Adv_{Kernel}^{Transient}$. Finally, after writing itself to filesystem, it becomes $Adv_{Kernel}^{Persist}$. We model this attack as a sequence of formulas,

$$\mathbf{E}\left(Adv_{Sandbox}^{Transient}, Browser\ Exploit\right) \implies Adv_{User}^{Transient}$$

$$\mathbf{E}\left(Adv_{User}^{Transient}, Kernel\ Exploit\right) \implies Adv_{Kernel}^{Transient}$$

$$\mathbf{E}\left(Adv_{Kernel}^{Transient}, Filesystem\ Write\right) \implies Adv_{Kernel}^{Persist}$$

In this example, the adversary's privilege escalation requires the exercise of three primitives: browser exploit, kernel exploit, and write to filesystem. Exercising these primitives escalates the adversary from $Adv_{Sandbox}^{Transient}$ to $Adv_{Kernel}^{Persist}$. A properly designed system should prevent the adversary from exercising these primitives. We define the security model in §4.

3.4. Adversary's Temporal Continuity

The actions of an adversary are rarely instantaneous, but rather occur over time. We must also model the temporal continuity of adversary capabilities. As an example, the recent self-propagating truck-to-truck malware [28] exploits the Electronic Logging Devices (ELDs), which are required by law to be installed on commercial trucks. The root cause of the attack is that the Over-The-Air (OTA) update interface is exposed to the trucks' WiFi proximity with a hardcoded password. The attack can be modeled as:

$$\left(Adv_{Proximity}^{Transient}\ \boldsymbol{U}\ \mathbf{E}\left(Adv_{Proximity}^{Transient}, OTA\ Update\right)\right) \implies Adv_{\text{ELD}}^{Persist}$$

3.5. Security Impacts

Traditionally, Confidentiality, Integrity, and Availability (the CIA triad) are the three primary security objectives. In our extended LTL notation, we use $\langle C, I, A\rangle$ to denote the consequences of an adversary's impacts on these objectives. The triad has been extended to include other objectives, such as non-repudiation and authenticity. These additional objectives can be added to the bracket notation as needed.

For example, the Meltdown and Spectre vulnerabilities [29], [30] let an unprivileged user-mode application (i.e., $Adv_{User}^{Transient}$), or a sandboxed JavaScript/eBPF program (i.e., $Adv_{Sandbox}^{Transient}$) trick CPU into Out-of-Order Execution (OOE), and use timing side-channels to infer the secrets of a victim application. Note that adversary's capabilities do not escalate in Meltdown and Spectre attacks unless they are combined with other exploits. However, the confidentiality of CPU, $\langle C\rangle_{Substrate}$, is breached transiently. The Meltdown and Spectre attacks can both be modeled as,

$$\mathbf{E}\left(Adv_{User}^{Transient} \vee Adv_{Sandbox}^{Transient}, \text{OOE}\right) \implies \langle C\rangle_{Substrate}^{Transient}$$

On top of Meltdown, the EchoLoad attack [31] further breaks Kernel Address Space Layout Randomization (KASLR). Compromising KASLR alone does not compromise a kernel's confidentiality (except for the kernel's base address), integrity, and availability, but it allows an adversary to launch a stronger exploit by knowing the kernel memory layout. Therefore, when KASLR is broken, the condition of another kernel exploit utilizing KASLR information is met. The EchoLoad attack can be modeled as,

$$\langle C\rangle_{KASLR}^{Transient} \implies \Diamond\mathbf{E}\Big(\ Adv_{User}^{Transient} \vee Adv_{Sandbox}^{Transient}, Other\ Kernel\ Exploit\ \Big)$$

Because KASLR resets after every reboot, the end result of attack is transient unless the adversary further writes malicious executable to filesystem. Putting together, the EchoLoad attack can be modeled as,

$$\Big(\ \mathbf{E}\Big(\ Adv_{User}^{Transient} \vee Adv_{Sandbox}^{Transient}, Meltdown\ \Big) \implies \langle C\rangle_{KASLR}^{Transient}\ \Big) \wedge \Big(\ \langle C\rangle_{KASLR}^{Transient} \implies \Diamond\mathbf{E}\left(Adv_{User}^{Transient} \vee Adv_{Sandbox}^{Transient}, Other\ Kernel\ Exploit\right)\ \Big) \wedge \Big(\ \mathbf{E}\Big(\ Adv_{User}^{Transient} \vee Adv_{Sandbox}^{Transient}, Other\ Kernel\ Exploit\ \Big) \implies \Big(\ \langle C, I, A\rangle_{Kernel}^{Transient}\ \Big)\ \Big)$$

4. Formal Trust Modeling

Corresponding to the threat modeling in §3, we define the necessary trust in system security guarantees. These guarantees are formulated as security objectives with goals to protect the $\langle C, I, A\rangle$ of system components.

System	Formula
Linux [32]	$\Box\Big(\ (\ \mathbf{E}(\ Adv^*_{User}, OS\ Bug\ )\vee\mathbf{E}(\ Adv^*_{User}, Dri\ Bug\ )\ )\implies\Big(\ \neg Adv^*_{Kern}\wedge\neg\langle C,I,A\rangle^*_{Dri,Kern}\ \Big)\Big)$
seL4 [33]	$\Box\mathbf{E}(\ Adv^*_{User}, OS\ Bug\ )\implies\Big(\ \neg Adv^*_{Kern}\wedge\neg\langle C,I,A\rangle^*_{Dri,Kern}\ \Big)$
Android [34]	$\Box(\ \mathbf{E}(\ Adv^*_{Sandbox}, OS\ Bug\ )\vee\mathbf{E}(\ Adv^*_{Sandbox}, Dri\ Bug\ )\ )\implies\Big(\ \neg Adv^*_{User}\wedge\neg Adv^*_{Kern}\wedge\neg\langle C,I,A\rangle^*_{Dri,Kern}\ \Big)$
Exokernel [35]	$\Box\mathbf{E}(\ Adv^*_{User}, Resource\ Hogging\ )\implies\bigcirc\neg\langle A\rangle^*_{SystemResource}$
Xen [36]	$\Box\Big(\ \mathbf{E}(\ Adv^*_{Kern}, VMM\ Bug\ )\implies\Big(\ \neg Adv^*_{Hypervisor}\wedge\neg\langle C,I,A\rangle^*_{VMM}\ \Big)\Big)$
AMD SEV [37]	$\Box\Big(\ \mathbf{E}\Big(\ Adv^*_{Hypervisor}, CPU\ Bug\ \Big)\implies(\ \neg Adv^*_{Hardware}\wedge\neg\langle C,I,A\rangle^*_{VM}\ )\Big)$
NoHype [38]	$\Box\Big(\ (\ \mathbf{E}(\ Adv^*_{VM}, CPU\ Bug\ )\vee\mathbf{E}(\ Adv^*_{VM}, I/O\ Bug\ )\ )\implies\Big(\ \neg Adv^*_{Hardware}\wedge\neg\langle C,I,A\rangle^*_{I/O\ Hardware}\ \Big)\Big)$
Arrakis [39]	$\Box(\ \mathbf{E}(\ Adv^*_{User}, OS\ Bug\ )\vee\mathbf{E}(\ Adv^*_{User}, I/O\ Bug\ )\ )\implies\Big(\ \neg Adv^*_{User}\wedge\neg Adv^*_{Kern}\wedge\neg\langle C,I,A\rangle^*_{Kern,\ I/O}\ \Big)$
Intel SGX [12]	$\Box(\ \mathbf{E}(\ Adv^*, OS^*\ )\implies(\ \neg AdvEnclave^*\wedge\neg\langle C,I,A\rangle\ Enclave\ Memory^*\ )\ )$
ARM CCA [9]	$\Box(\ \mathbf{E}(\ Adv^*, Hypervisor^*\ )\implies(\ \neg AdvRealm^*\wedge\neg\langle C,I\rangle\ Realm\ Memory,\ Realm\ CPU\ State^*\ )\ )$
Keystone [40]	$\Box(\ \mathbf{E}(\ Adv^*, OS^*\ )\implies(\ \neg AdvEnclave^*\wedge\neg\langle C,I\rangle\ Enclave\ Memory^*\ )\ )$
CURE [41]	$\Box(\ \mathbf{E}(\ Adv^*, OS^*, DMA^*\ )\implies(\ \neg AdvEnclave^*\wedge\neg\langle C,I\rangle\ Enclave\ Memory^*\ )\ )$
MI6 [42]	$\Box(\ \mathbf{E}(\ Adv^*, OS^*, Hypervisor^*\ )\implies(\ \neg AdvEnclave^*\wedge\neg\langle C,I\rangle\ Enclave\ Memory,\ Enclave\ CPU\ State^*\ )\ )$
IRONHIDE [43]	$\Box(\ \mathbf{E}(\ Adv^*, OS^*\ )\implies(\ \neg AdvEnclave^*\wedge\neg\langle C,I,A\rangle\ Enclave\ Memory^*\ )\ )$
ProvCam [44]	$\Box(\ \mathbf{E}(\ Adv, OS, Hypervisor\ )\implies(\ \neg AdvCamera\ Module^*\wedge\neg\langle I\rangle\ Video^*\ )\ )$
CODOMs [45]	$\Box(\ \mathbf{E}(\ Adv, Hypervisor\ )\implies\neg\langle C,I\rangle^*_{Enclave}\ )$
CHERI [46]	$\Box\Big(\ \mathbf{E}(\ Adv^*, OS^*\ )\implies\neg\langle C,I\rangle^*_{Duo\ Memory}\ \Big)$

TABLE 1: Examples of trust and threat modeling formulas for training the LLM (in white background), and the results generated by the fine-tuned LLM (in yellow background). Kern denotes kernel, Dri denotes device driver. $*$ denotes any.

4.1. Security Guarantees

Trusting a system to have certain properties does not imply that the system is free of vulnerabilities. Indeed, systems often fail to meet their security guarantees [29], [30], [47]. Trust in a system is subjective, but we aim to formalize security guarantees to reduce the trust assumptions to a minimum. Modeling security guarantees also facilitates the verification of their correctness in design and implementation. We present eBPF runtime's security guarantees as an example.

eBPF is a kernel feature that allows user-space programs to deploy verified code snippets to run with an interpreter in the kernel-mode. eBPF verifier [48] traverses the code snippets for checking program states and behaviors [49]. $Adv^{Transient}_{eBPF}$ is a presumably malicious eBPF program that aims to exploit the verifier bug to inject malicious code into the kernel. After validation, eBPF runtime [50] runs the program and ensures it only accesses allowed kernel resources. eBPF maps are key-value stores that can be accessed by both the user-space and eBPF program. One important guarantee provided by the runtime is that the allocation patterns of these maps are random to prevent adversaries from inferring kernel heap layout and launching out-of-bound memory access attacks. $Executed\,(eBPF\ program)$ denotes an eBPF program executed by the runtime. eBPF runtime can be modeled as,

$$\begin{aligned}\Box\Big(\ \mathbf{E}\Big(\ Adv^{Transient}_{eBPF}, eBPF\ Exploit\ \Big)\\ \implies(\ \neg Adv^*_{Kernel}\wedge\neg Adv^*_{UserRoot}\ )\wedge\\ \mathbf{E}\Big(\ Adv^{Transient}_{eBPF}, Memory\ Probe\ \Big)\\ \implies\Diamond\neg\langle C\rangle^{Transient}_{Kernel}\wedge\\ \mathbf{E}\Big(\ Adv^{Transient}_{eBPF}, Illegal\ Memory\ Access\ \Big)\\ \implies\bigcirc\neg Executed\Big(\ Adv^{Transient}_{eBPF}\ \Big)\ \Big)\end{aligned}$$

which denotes that the adversary must not be able to

escalate its privilege to Adv_{Kernel} or $Adv_{UserRoot}$ through exploiting the eBPF runtime, the runtime must always prevent the adversary from probing kernel memory layout, and the runtime will halt illegal memory access instructions.

4.2. Description of Trust Levels

Trust level describes the trustworthiness of a system in fulfilling its security guarantees. A botched implementation of a complex system may not meet its claimed security guarantees, and hence, the trust level of such a system is low. In contrast, a well-reasoned design and well-tested implementation have a higher chance of meeting its guarantees, and deserve more trust. A formally verified system has the highest trust level, as its security guarantees are proven to be met. However, even a formally verified system may fail if the system's premises are violated, or if the system's specifications are not sound. As trust is subjective, we do not quantify the degree of trust. We model trust as a binary value. The notation of $\top$ and $\neg\top$ in the front of a security guarantee formula can be used to indicate whether the formula is trusted or not.

Interestingly, not all *trust* carries the same meaning. In the context of an operating system that only serves a single program, we only need to trust the operating system to operate correctly without any adversarial interference [15]. However, in the context of a general-purpose operating system that serves multiple programs, we need to trust the operating system to operate correctly, and to protect the $\langle C, I, A\rangle$ of each program and the operating system itself. We adopt the trust model originally proposed in Yao et al. [51]: $\top_{Resilient}$ denotes the system is trusted to be resilient to known and unknown attacks, and $\top_{Correct}$ denotes the system is trusted to operate correctly. $\top_{Correct}$ is a precondition for $\top_{Resilient}$. Further, superscripts can be used to indicate specific security objectives, such as $\top^{C}$, $\top^{I}$, and $\top^{A}$.

As an example, the trust in eBPF runtime, which must be resilient against $Adv_{eBPF}^{Transient}$, can be represented as, $\top_{Resilient}^{C,I,A}$. In contrast, in the case of an encryption algorithm, we only need to trust it to operate correctly to protect data confidentiality. This trust can be represented as, $\top_{Correct}^{C}$.

5. Methodology and Training Data

We fine-tune the GPT-4o model with our extended LTL annotation and our manually created security models of real-world systems, including the eBPF runtime (§4), the Linux kernel [32], seL4 [33], Android [34], Exokernel [35], Xen [36], and AMD SEV [37], NoHyper [38], and Arrakis [39]. The white rows in Table 1 show our manually written security model formulas for these examples. The selection of the systems is based on the following criteria: (1) The examples are from recent system security published at top-tier conferences and journals, as well as from well-known real-world systems. (2) The examples have clear assumptions on the adversaries' capabilities and the necessary security guarantees. (3) Their security models are unique and not covered by other examples. The training is on top of the latest OpenAI GPT-4o model (labeled gpt-4o-2024-08-06) as of the time of writing.

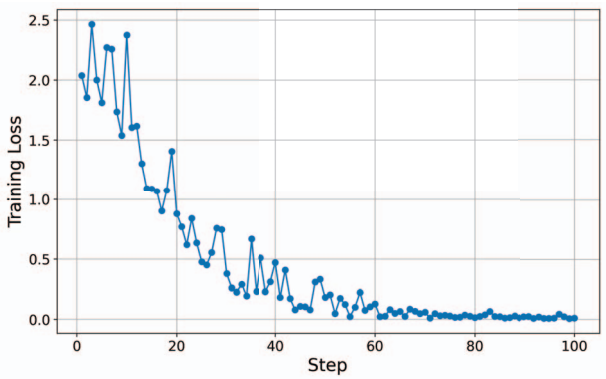

Figure 1: Training loss of GPT-4o fine-tuned with our security modeling examples, showing the loss converges effectively.

For each work, we train the LLM with system, user, and assistant role corresponding to the GPT-4o's training data format, and submit them to the fine-tuning interface. The system role is a brief description of our modeling notations, and it remains the same across all examples. Then, we supply a brief description of the system. For research papers, we train with the paper's abstract and not the full paper due to limitations on the training context size (see §8). For real-world systems, we train with a summary from reputable Internet sources as cited in the table. The fine-tuned model then takes (1) the system role prompt (a brief description of our modeling notations) and (2) a computer system's description in natural language as input, and outputs the security model of the system in LTL formulas.

6. Evaluation

We conduct fine-tuning with GPT-4o with 37,130 training tokens of 10 modeling formulas as listed in §5. GPT-4o finishes training in 100 steps (10 epochs) with a batch size of 1 and a final training loss of 0.0096. We show the training loss over the 100 steps in Figure 1, which demonstrates that the model converges effectively.

We evaluate our fine-tuned model with a set of 9 systems. Although the fine-tune interface has limitations on the training token size, the inference of the model allows for the evaluation of the entire research papers. In our experiment, we use the same system role prompt as in our training data to give a brief description of our extended LTL logic annotations, and provide a one-sentence user role prompt, "Conclude the trust and threat model of <*system name*>" to the model. We attach the research papers to the fine-tuned model through OpenAI's web user interface. The correctness of modeling is evaluated by manually comparing the generated LTL formulas with the descriptions of the systems. In all the cases, the model is able to retrieve the

document, and provide a detailed modeling of the system (as shown in the yellow rows in Table 1).

For example, the model recognizes SGX guarantees the $\langle C, I, A\rangle\, Enclave\ Memory^*$ (the confidential, integrity, and availability of enclave memory) in the presence of an adversary that exploits operating system, and the adversary cannot escalate to the enclave. In comparison, ARM CCA is recognized to guarantee the $\langle C, I\rangle\, Realm\ Memory,\ Realm\ CPU\ State^*$ in the presence of an adversary that exploits hypervisor, and the adversary cannot escalate to the Realm. For ProvCam [44], the model correctly indicates that the system protects the $\langle I\rangle\, Video^*$ (integrity of video) and the adversary cannot escalate to the camera module hardware. For Keystone [40], CURE [41], MI6 [42], ARM CCA [9], CODOMs [45], and CHERI [46], the model correctly identifies that these works do not explicitly guarantee the availability of system. Overall, LLM correctly models these systems in our extended LTL logic formulas. The only deficiency is that LLM misses the annotation of adversaries' prepense levels (e.g., $Adv^*_{Sandbox}$), which is a minor issue because the adversaries' prepense level are commonly assumed to be one level lower than the exploited system's role. By explicitly emphasizing this annotation in the fine-tuning process and providing more training data, we believe that the model can be further improved to mitigate this issue.

7. Related Work

7.1. Security Modeling

Security modeling is a well-established field in research. Swiderski et al. [25], Sanzgiri et al. [20], and Van Landuyt et al. [52] proposed data-based dynamic threat modeling methods. Alecci et al. [53] introduce a formal modeling approach for adversarial transferability analysis. Xiong et al. [54] conducted a systematic literature review on threat modeling. These works are orthogonal to our work, as their goals are to model the threats in a system, while our work focuses on modeling the trust assumptions and threats of a system with the help of LLMs.

7.2. AI Security

AI models have been used in various system security and privacy applications, such as vehicle safety [55], [56], software security [57], [58], anomaly detection [59], [60], and hardware security [61]. Also, AI systems themselves have been shown to be vulnerable to security and privacy attacks [62]. Prompt filtering and sanitization prevent privacy hazards in using cloud-based LLMs [63]–[65]. Dev et al. [66] introduced guardrails with AI/ML threat analysis. These works focus on AI security, while our work highlights the use of AI in security modeling.

7.3. LLM-based Threat Modeling

LLM has been used in modeling system and network security. PILLAR [67] utilizes LLM to convert natural language privacy descriptions to privacy models. Elsharef et al. [68] use LLM to create threat modeling questions throughout the software development lifecycle. Moskal et al. [69] present a study on the use of LLM in network threat modeling. These works are complementary to ours, as they focus on using LLMs to model various aspects of security, while our work focuses on using LLMs to formally model trust and threat.

8. Discussion and Future Work

As LLM fine-tuning context window is expected to be improved, we expect that in the future, we may be able to train with more detailed descriptions, such as the full paper or the system's documentation. However, we find that before fine-tuning, LLM is already aware of all the systems used in our training in Table 1, which is a good sign that LLM has a good background knowledge of well-known systems.

We acknowledge that the training data is not exhaustive due to the limitation of manually deriving the modelings, and we envision that a reinforcement learning approach can be used to automatically generate more examples for training. Recent research has shown iterative prompt refinement can improve the output quality of LLM [70], [71]. A promising direction is to iteratively include previously generated modelings to refine the prompt for the next generation of LLMs. An important task is to dynamically evaluate the correctness and completeness of the generated modelings, which is left for future work.

In addition, we understand a system may have multiple components, where each component may have different trust and threat models. These models may also evolve dynamically as the system is in use. We envision that Computation Tree Logic (CTL) [72] can be used to model a hierarchical model, and the LLM-based modeling can be updated dynamically as the system states evolve.

9. Conclusion

In this work, we introduce an extended LTL annotation for accurate trust and threat modeling of a system, and integrate the formal modeling with a fine-tuned LLM. We manually write 10 formulas for the security models of real-world systems and attack scenarios, and fine-tune GPT-4o with them. We evaluate the fine-tuned model with a set of 9 system security models to show the model's capability in capturing their security models. Our method addresses the limitations of traditional security modeling that relies on imprecise natural language that often leads to inconsistencies, and it highlights the benefits of using LLMs for security modeling.

Acknowledgments

This work was supported in part by the New Jersey Institute of Technology new faculty startup fund. The author thanks OpenAI for donating API credits to this project.

References

[1] J. Vander Stoep, "Android: Protecting the Kernel," in *Linux Security Summit (LSS)*, 2016.

[2] S. M. Seyed Talebi, Z. Yao, A. Amiri Sani, Z. Qian, and D. Austin, "Undo Workarounds for Kernel Bugs," in *Proc. USENIX Security Symposium*, 2021.

[3] Eclypsium, "Screwed Drivers - Signed, Sealed, Delivered," https://eclypsium.com/research/screwed-drivers-signed-sealed-delivered/, 2019.

[4] Google, "The Chromium Chronicle: Coding Outside the Sandbox," https://developers.google.com/web/updates/2019/08/chromium-chronicle-5, 2019.

[5] Z. Yao, Z. Ma, Y. Liu, A. Amiri Sani, and A. Chandramowlishwaran, "Sugar: Secure GPU Acceleration in Web Browsers," in *Proc. ACM ASPLOS*, 2018.

[6] Z. Yao, S. Mirzamohammadi, A. Amiri Sani, and M. Payer, "Milkomeda: Safeguarding the Mobile GPU Interface Using WebGL Security Checks," in *Proc. ACM CCS*, 2018.

[7] H. Peng, Z. Yao, A. Amiri Sani, D. J. Tian, and M. Payer, "GLeeFuzz: Fuzzing WebGL Through Error Message Guided Mutation," in *Proc. USENIX Security Symposium*, 2023.

[8] R. Grisenthwaite, "Arm CCA will put confidential compute in the hands of every developer," https://www.arm.com/company/news/2021/06/arm-cca-will-put-confidential-compute-in-the-hands-of-every-developer, 2021.

[9] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell, "Design and verification of the arm confidential compute architecture," in *Proc. USENIX OSDI*, 2022.

[10] ARM, "Arm Security Technology Building a Secure System using TrustZone Technology," http://infocenter.arm.com/, 2009.

[11] Intel, "Innovative Instructions and Software Model for Isolated Execution," https://software.intel.com/content/www/us/en/develop/articles/innovative-instructions-and-software-model-for-isolated-execution.html, 2013.

[12] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Proc. Inter. Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[13] A. Athalye, A. Belay, M. Kaashoek, R. Morris, and N. Zeldovich, "Notary: A device for secure transaction approval," in *Proc. ACM SOSP*, 2019.

[14] A. Amiri Sani and T. Anderson, "The Case for I/O-Device-as-a-Service," in *Proc. ACM HotOS*, 2019.

[15] Z. Yao, S. M. Seyed Talebi, M. Chen, A. Amiri Sani, and T. Anderson, "Minimizing a smartphone's tcb for security-critical programs with exclusively-used, physically-isolated, statically-partitioned hardware," in *Proc. ACM MobiSys*, 2023.

[16] D. Zueck, N. Atallah, I. Do, Z. Yao, and A. Amiri Sani, "Hora: High assurance periodic availability guarantee for life-critical applications on smartphones," in *Proc. ACM APSYS*, 2024.

[17] J. H. Moreno, H. Franke, P. Crumley, and M. Ye, "Calling for the Return of Non-Virtualized Microprocessor Systems," https://www.sigarch.org/calling-for-the-return-of-non-virtualized-microprocessor-systems/, 2022.

[18] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proc. IEEE/ACM ICSE*, 1999.

[19] J. Mickens, "This world of ours," *USENIX ;login:*, 2014.

[20] A. M. Sanzgiri and S. J. Upadhyaya, "Feasibility of attacks: What is possible in the real world-a framework for threat modeling," in *Proceedings of the International Conference on Security and Management (SAM)*, 2011.

[21] M. Barbaro and T. Zeller, "A Face Is Exposed for AOL Searcher No. 4417749," https://www.nytimes.com/2006/08/09/technology/09aol.html, 2006.

[22] N. Perlroth, "All 3 Billion Yahoo Accounts Were Affected by 2013 Attack," https://www.nytimes.com/2017/10/03/technology/yahoo-hack-3-billion-users.html, 2017.

[23] K. Conger and S. Frenkel, "Thousands of Microsoft Customers May Have Been Victims of Hack Tied to China," https://www.nytimes.com/2021/03/06/technology/microsoft-hack-china.html, 2021.

[24] E. Barrett, "A hack at Equifax exposed the data of 147 million people. Here's what businesses can learn from the company's response," https://fortune.com/2023/08/18/lessons-from-equifax-security-breach, 2017.

[25] F. Swiderski and W. Snyder, *Threat modeling*. Microsoft Press, 2004.

[26] A. Pnueli, "The temporal logic of programs," in *Proc. IEEE SFCS*, 1977.

[27] "Spyware vendors use 0-days and n-days against popular platforms," https://blog.google/threat-analysis-group/spyware-vendors-use-0-days-and-n-days-against-popular-platforms, 2023.

[28] J. Jepson, R. Chatterjee, and J. Daily, "Commercial vehicle electronic logging device security: Unmasking the risk of truck-to-truck cyber worms," in *Proc. Symposium on Vehicle Security and Privacy*, 2024.

[29] 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 *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2019.

[30] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *Proc. USENIX Security Symposium*, 2018.

[31] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "Kaslr: Break it, fix it, repeat," in *Proc. ACM Asia CCS*, 2020.

[32] "The Linux Kernel Archives," https://www.kernel.org/, 2024.

[33] 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," in *Proc. ACM SOSP*, 2009.

[34] "Android Open Source Project," https://source.android.com/, 2024.

[35] D. R. Engler, M. F. Kaashoek, and J. O. Jr., "Exokernel: an Operating System Architecture for Application-Level Resource Management," in *Proc. ACM SOSP*, 1995.

[36] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proc. ACM SOSP*, 2003.

[37] "AMD Secure Encrypted Virtualization (SEV)," https://www.amd.com/en/developer/sev.html, 2024.

[38] E. Keller, J. Szefer, J. Rexford, and R. Lee, "Nohype: virtualized cloud infrastructure without the virtualization," in *Proc. IEEE/ACM ISCA*, 2010.

[39] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The Operating System is the Control Plane," in *Proc. USENIX OSDI*, 2014.

[40] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An Open Framework for Architecting Trusted Execution Environments," in *Proc. ACM EuroSys*, 2020.

[41] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A. Sadeghi, and E. Stapf, "CURE: A Security Architecture with CUstomizable and Resilient Enclaves," in *Proc. USENIX Security Symposium*, 2021.

[42] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *Proc. ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2019.

[43] H. Omar and O. Khan, "IRONHIDE: A Secure Multicore that Efficiently Mitigates Microarchitecture State Attacks for Interactive Applications," in *Proc. IEEE HPCA*, 2020.

[44] Y. Liu, Z. Yao, M. Chen, A. Amiri Sani, S. Agarwal, and G. Tsudik, "Provcam: A camera module with self-contained tcb for producing verifiable videos," in *Proc. ACM MobiCom*, 2024.

[45] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero, "CODOMs: Protecting Software with Code-centric Memory Domains," in *Proc. ACM/IEEE ISCA*, 2014.

[46] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI Capability Model: Revisiting RISC in an Age of Risk," in *Proc. ACM/IEEE ISCA*, 2014.

[47] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *Proc. USENIX Security Symposium*, 2016.

[48] "eBPF Verifier," https://www.kernel.org/doc/html/latest/bpf/verifier.html, 2024.

[49] H. Sun and Z. Su, "Validating the {eBPF} verifier via state embedding," in *Proc. USENIX OSDI*, 2024.

[50] "eBPF Major Infrastructure," https://ebpf.io/infrastructure, 2024.

[51] Z. Yao, S. M. Seyed Talebi, M. Chen, A. Amiri Sani, and T. Anderson, "Minimizing trust with exclusively-used physically-isolated hardware," *arXiv preprint arXiv:2203.08284*, 2022.

[52] D. Van Landuyt, L. Pasquale, L. Sion, and W. Joosen, "Threat modeling at run time: the case for reflective and adaptive threat management (nier track)," in *IEEE SEAMS*, 2021.

[53] M. Alecci, M. Conti, F. Marchiori, L. Martinelli, and L. Pajola, "Your attack is too dumb: Formalizing attacker scenarios for adversarial transferability," in *ACM RAID*, 2023.

[54] W. Xiong and R. Lagerström, "Threat modeling-a systematic literature review," *Computers & security*, vol. 84, pp. 53–69, 2019.

[55] O. Adeboye, T. Dargahi, M. Babaie, M. Saraee, and C. Yu, "Deepclean: a robust deep learning technique for autonomous vehicle camera data privacy," *IEEE Access*, vol. 10, pp. 124 534–124 544, 2022.

[56] B. Desai and K. Patil, "Secure and scalable multi-modal vehicle systems: A cloud-based framework for real-time llm-driven interactions," *Innovative Computer Sciences Journal*, vol. 9, no. 1, pp. 1–11, 2023.

[57] Y. Jiang, J. Liang, F. Ma, Y. Chen, C. Zhou, Y. Shen, Z. Wu, J. Fu, M. Wang, S. Li *et al.*, "When fuzzing meets llms: Challenges and opportunities," in *Proc. ACM FSE*, 2024.

[58] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proc. Internet Society NDSS*, 2024.

[59] X. Wang, Z. Yao, and M. Papaefthymiou, "A real-time electrical load forecasting and unsupervised anomaly detection framework," *Applied Energy*, vol. 330, p. 120279, 2023.

[60] M. Zolanvari, A. Ghubaish, and R. Jain, "Addai: Anomaly detection using distributed ai," in *Proc. IEEE ICNSC*, 2021.

[61] D. Saha, S. Tarek, K. Yahyaei, S. K. Saha, J. Zhou, M. Tehranipoor, and F. Farahmandi, "Llm for soc security: A paradigm shift," *IEEE Access*, 2024.

[62] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[63] C. J. Chong, C. Hou, Z. Yao, and S. M. Seyed Talebi, "Casper: Prompt sanitization for protecting user privacy in web-based large language models," *arXiv preprint arXiv:2408.07004*, 2024.

[64] X. Yue, M. Du, T. Wang, Y. Li, H. Sun, and S. S. M. Chow, "Differential privacy for text analytics via natural text sanitization," in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, 2021, pp. 3853–3866.

[65] Z. Shen, Z. Xi, Y. He, W. Tong, J. Hua, and S. Zhong, "The fire thief is also the keeper: Balancing usability and privacy in prompts," *arXiv preprint arXiv:2406.14318*, 2024.

[66] J. Dev, N. Akhuseyinoglu, G. Kayas, B. Rashidi, and V. Garg, "Building guardrails in ai systems with threat modeling," *Digital Government: Research and Practice*, 2024.

[67] M. Mollaeefar, A. Bissoli, and S. Ranise, "Pillar: an ai-powered privacy threat modeling tool," *arXiv preprint arXiv:2410.08755*, 2024.

[68] I. Elsharef, Z. Zeng, and Z. Gu, "Facilitating threat modeling by leveraging large language models," in *Proc. Workshop on AI Systems with Confidential Computing (AISCC)*, 2024.

[69] S. Moskal, S. Laney, E. Hemberg, and U.-M. O'Reilly, "Llms killed the script kiddie: How agents supported by large language models change the landscape of network threat testing," *arXiv preprint arXiv:2310.06936*, 2023.

[70] N. McAleese, R. M. Pokorny, J. F. C. Uribe, E. Nitishinskaya, M. Trebacz, and J. Leike, "Llm critics help catch llm bugs," *arXiv preprint arXiv:2407.00215*, 2024.

[71] C. J. Chong, Z. Yao, and I. Neamtiu, "Artificial-intelligence generated code considered harmful: A road map for secure and high-quality code generation," *arXiv preprint arXiv:2409.19182*, 2024.

[72] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on logic of programs*. Springer, 1981, pp. 52–71.