IDS4.0 测试结果报告

测试类名: TestAlert

测试方法名	描述	预期结果	实际结果
test_alert_creation	测试 Alert 创建	告警功能应正常工作	测试通过
test_alert_str	测试 Alert 对象的字符串表示功能	1. 字符串应包含严重程度信息（CRITICAL）; 2. 字符串应包含告警类型（test_alert）; 3. 字符串应包含 CAN ID（123）; 4. 字符串应包含详细信息（Test details）	测试通过
test_alert_to_dict	测试 Alert 对象转换为字典格式功能	1. 字典应包含 alert_type、can_id、details 等基本字段; 2. timestamp 字段应保持原始值; 3. severity 字段应转换为小写字符串; 4.应自动生成唯一的 alert_id 字段	测试通过

测试类名: TestAlertManager

测试方法名	描述	预期结果	实际结果
test_alert_throttling_cooldown	验证 AlertManager 的全局冷却时间机制能够正确限制 在冷却期内的所有告警，无论 CAN ID 是否相同。	1. 发送第一个告警应成功处理; 2. 在 100ms 后发送第二个告警(小于 250ms 冷却时间); 3. 第二个告警应被限流，不被处理; 4. 总告警数(total_alerts)应为 1; 5. 被限流的告警数(throttled_alerts)应大于 0; 6. 冷却机制应正确工作	测试通过
test_alert_throttling_per_id	验证 AlertManager 的限流机制能够正确限制同一 CAN ID 在短时间内 产生的告警数量，防止告警洪水攻击。	1. 发送 5 个相同 ID 的告警，间隔 0.1 秒; 2. 由于冷却时间 250ms 的限制，只有前 2 个告警应被处理; 3. 总告警数	测试通过

		(total_alerts)应为 2; 4. 被限流的告警数(throttled_alerts)应大于 0; 5. 限流机制应有效防止告警泛滥	
test_clear_statistics	验证 AlertManager 能够正确清除所有统计信息和历史记录， 将系统重置到初始状态。	1. 发送一个测试告警后，统计信息应存在; 2. 手动清除统计信息后:; - 总告警数(total_alerts)应重置为 0; - 按类型统计字典应为空; - 最近告警列表应为空; - 最后告警时间戳应重置为 0.0; 3. 系统应完全重置到初始状态	测试通过
test_file_output_content	验证 AlertManager 能够正确将告警信息写入到日志文件和 JSON 文件中， 确保文件内容的完整性和正确性。	1. 发送测试告警后， alerts.log 文件应存在; 2. alerts.json 文件应存在; 3. 两个文件的大小应大于等于 0(表示有内容写入); 4. 文件缓冲区应正确刷新; 5. 文件内容应包含告警的相关信息; 6. 文件输出功能应正常工作	测试通过
test_get_statistics	验证 AlertManager 能够正确收集和返回各种告警统计信息， 包括总数、按类型分组、按严重程度分组等。	1. 发送 3 个不同的告警，间隔大于冷却时间; 2. 总告警数(total_alerts)应为 3; 3. 按类型统计: test1 类型应为 2 个，test2 类型应为 1 个; 4. 按严重程度统计: high 应为 1 个，medium 应为 1 个，low 应为 1 个; 5. 统计信息应准确反映所有已处理的告警	测试通过
test_initialization	验证 AlertManager 实例能够正确初始化，包括基本属性设置、统计信息初始化、 输出目录创建和输出配置设置。	1. output_dir 属性应等于传入的临时目录路径; 2. config_manager 属性应等于传入的模拟配置管理器; 3. alert_stats['total_alerts']应初始化为 0; 4. alert_stats['alerts_by_type']应为 defaultdict 类型; 5. 输出目录应成功创建并存在; 6. 控制台输出配置	测试通过

		应启用(enabled=True); 7. 文件输出配置应启用(enabled=True); 8. JSON 文件输出配置应启用(enabled=True)	
test_invalid_alert_handling	验证 AlertManager 能够优雅地处理无效的告警数据， 不会导致系统崩溃或异常。	1. 发送无效的告警数据(字符串类型)不应导致异常; 2. 系统应优雅地忽略无效告警; 3. 统计信息不应因无效告警而更新; 4. 总告警数(total_alerts)应保持为 0; 5. 系统应具有良好的错误容错能力	测试通过
test_output_configuration	验证 AlertManager 能够正确处理不同的输出配置选项， 包括禁用控制台输出和更改输出格式。	1. 禁用控制台输出后， 系统应正常工作; 2. 更改文件输出格式为 JSON 后， 系统应正常工作; 3. 发送测试告警不应导致任何异常; 4. 配置更改应不影响告警处理的核心功能; 5. 系统应具有良好的配置灵活性	测试通过
test_output_files_creation	验证 AlertManager 能够正确创建告警日志文件和 JSON 文件， 并确保文件句柄正确初始化。	1. alerts.log 文件应在输出目录中成功创建; 2. alerts.json 文件应在输出目录中成功创建; 3. _alert_file 文件句柄应不为 None; 4. _json_file 文件句柄应不为 None; 5. 两个文件都应该可以正常写入	测试通过
test_report_alert_with_alert_object	验证 AlertManager 能够正确处理 Alert 对象类型的告警， 包括统计信息更新和告警历史记录。	1. 总告警数(total_alerts)应增加到 1; 2. 按类型统计(alerts_by_type['test_alert'])应为 1; 3. 按 ID 统计(alerts_by_id['0x123'])应为 1; 4. 按严重程度统计(alerts_by_severity['high'])应为 1; 5. 最近告警列表(recent_alerts)长度应为 1; 6. 最近告警列表中的第一个元素应等于报告的告警对象	测试通过

test_report_alert_with_dict	验证 AlertManager 能够正确处理字典类型的告警数据， 并正确更新各项统计信息。	1. 总告警数(total_alerts)应增加到 1; 2. 按类型统计(alerts_by_type['dict_alert'])应为 1; 3. 按 ID 统计(alerts_by_id['0x456'])应为 1; 4. 按严重程度统计(alerts_by_severity['medium'])应为 1; 5. 字典格式的告警应被正确解析和处理	测试通过

测试类名: TestBaseDetector

测试方法名	描述	预期结果	实际结果
test_config_cache_cleanup	验证 BaseDetector 能够正确清理过期的配置缓存， 确保内存使用合理，避免缓存数据过度积累。	1. 配置缓存清理方法应成功执行; 2. 过期的缓存条目应被正确清除; 3. 有效的缓存条目应保留; 4. 清理后内存使用应减少; 5. 清理过程不应影响正常的配置访问; 6. 系统性能应得到优化	测试通过
test_config_version_change	验证 BaseDetector 能够正确检测配置版本的变化， 并在配置更新时清理旧的缓存， 重新加载新配置。	1. 配置版本变化检测应正常工作; 2. 检测到版本变化时，应清理旧的配置缓存; 3. 新配置应正确加载到缓存中; 4. 配置更新过程不应影响检测器正常运行; 5. 版本变化响应机制应及时有效; 6. 系统应能适应动态配置更新	测试通过
test_create_alert	验证 BaseDetector 基类的 _create_alert 方法能够正确创建 Alert 对象， 并自动设置检测上下文信息，包括检测器名称和检测时间。	1. _create_alert 方法应成功创建 Alert 对象; 2. 创建的 Alert 对象类型应为 Alert 类实例; 3. Alert 的 alert_type 应等于传入的 "test_alert"; 4. Alert 的 can_id 应等于帧的 CAN ID; 5. Alert 的 details 应等于传入的详细信息; 6.	测试通过

		Alert 的 severity 应等于传入的严重程度; 7. Alert 的 detection_context 应包含检测器名称; 8. Alert 的 detection_context 应包含检测时间信息	
test_detect_method	验证具体检测器实现的 detect 方法能够正确分析 CAN 帧数据，并在检测到异常时生成相应的告警对象。	1. detect 方法应成功执行，返回告警列表; 2. 当 CAN ID 为 "alert_test"时，应生成一个告警; 3. 生成的告警类型应为 "test_alert"; 4. 告警的严重程度应为 AlertSeverity.HIGH; 5. 告警应包含正确的 CAN 帧信息; 6. 当 CAN ID 不匹配时，应返回空的告警列表; 7. 检测逻辑应正确执行，不抛出异常	测试通过
test_detector_initialization	测试 BaseDetector 检测器初始化功能	1. config_manager 属性应正确设置; 2. _config_cache 应初始化为空字典; 3. detector_name 应设置为类名; 4. 检测器应处于可用状态	测试通过
test_get_cached_config	验证 BaseDetector 能够正确缓存和获取配置信息，提高配置访问性能，避免重复的配置读取操作。	1. get_cached_config 方法应成功执行; 2. 首次调用应从配置管理器读取配置; 3. 后续调用应返回缓存的配置，提高性能; 4. 缓存的配置内容应与原始配置一致; 5. 配置缓存机制应正常工作; 6. 不应因缓存导致配置信息错误	测试通过
test_get_config_value	验证 BaseDetector 能够正确获取指定的配置值，包括处理默认值、类型转换和异常情况。	1. get_config_value 方法应成功执行; 2. 存在的配置项应返回正确的值; 3. 不存在的配置项应返回指定的默认值; 4. 配置值类型应正确转换; 5. 异常情况应得到妥善处理; 6. 配置访问应高效可靠	测试通过
test_is_detection_enabled	验证 BaseDetector 能够正确判断	1. is_detection_enabled 方法应成	测试通过

	检测功能是否启用， 通过配置管理器获取相应的配置值并返回正确的布尔结果。	功执行; 2. 当配置中检测功能启用时， 应返回 True; 3. 当配置中检测功能禁用时， 应返回 False; 4. 方法应正确调用配置管理器获取配置值; 5. 返回值应为布尔类型; 6. 配置读取过程不应抛出异常	
test_memory_pressure_mode	验证 BaseDetector 在系统内存压力较大时能够正确调整行为， 包括减少缓存使用、优化内存分配等内存管理策略。	1. 内存压力模式应正确启用; 2. 检测器应减少内存使用， 清理非必要缓存; 3. 核心检测功能应继续正常工作; 4. 内存使用应得到有效控制; 5. 系统稳定性应得到保障; 6. 性能降级应在可接受范围内	测试通过
test_post_detect	验证 BaseDetector 的 post_detect 方法能够正确执行检测后的清理工作， 包括统计信息更新、资源清理等后处理步骤。	1. post_detect 方法应成功执行，不抛出异常; 2. 方法应正确接收 alerts 和 detection_time 参数; 3. 统计信息应正确更新(如 detection_count); 4. 如果有告警， alert_count 应相应增加; 5. last_detection_time 应更新为最新的检测时间; 6. 后处理逻辑应正确执行， 不影响系统稳定性	测试通过
test_pre_detect	验证 BaseDetector 的 pre_detect 方法能够正确执行检测前的准备工作， 包括参数验证、状态检查等预处理步骤。	1. pre_detect 方法应成功执行，不抛出异常; 2. 方法应正确接收 frame、id_state 和 config_proxy 参数; 3. 预处理逻辑应正确执行; 4. 方法执行后系统状态应保持正常; 5. 为后续的 detect 方法调用做好准备; 6. 不应影响检测器的正常功能	测试通过

测试类名: TestBaselineEngine

测试方法名	描述	预期结果	实际结果
test_complete_learning	验证 BaselineEngine 能够正确完	1. is_learning_active 应设置为	测试通过

	成学习过程，包括生成基线数据、更新学习状态和返回完整的基线信息。	False; 2. learning_completed 应设置为 True; 3. 应返回包含基线数据的字典; 4. 基线数据应包含"0x123"的 ID 信息; 5. ID 基线应包含 frame_count、first_seen、last_seen 等字段; 6. 基线数据应完整准确，可用于后续检测; 7. 学习状态应正确转换为完成状态	
test_get_baseline_for_id	验证 BaselineEngine 能够正确返回指定 CAN ID 的基线数据，包括存在的 ID 和不存在的 ID 的处理。	1. 存在的 ID "0x123"应返回非空的基线数据; 2. 存在的 ID "0x456"应返回非空的基线数据; 3. 不存在的 ID "0x999"应返回 None; 4. 返回的基线数据应包含 frame_count 字段; 5. ID "0x123"的 frame_count 应为 1; 6. ID "0x456"的 frame_count 应为 1; 7. 基线数据应包含正确的 DLC 信息(8 和 4)	测试通过
test_get_learning_statistics	验证 BaselineEngine 能够正确生成和返回学习过程的统计信息，包括学习的 ID 数量、帧数量、持续时间和状态等。	1. 返回的统计信息应为字典类型; 2. 应包含 learned_id_count 字段，值为 2; 3. 应包含 total_frame_count 字段，值为 3; 4. 应包含 total_duration 字段; 5. 应包含 learning_status 字段，值为'active'; 6. 统计信息应准确反映当前学习状态; 7. 所有统计数据应与实际处理的数据一致	测试通过
test_initialization	验证 BaselineEngine 能够正确初始化，包括配置管理器设置、学习参数配置、状态变量初始化和数据结构准备等基本功能。	1. config_manager 属性应正确设置为传入的配置管理器; 2. learning_duration 应从配置中正确读取为 30 秒; 3. min_samples 应从配置中正确读取为 100; 4. is_learning_active 应初始化为 False; 5. learning_completed 应初	测试通过

		始化为 False; 6. learning_start_time 应初始化为 None; 7. data_per_id 应初始化为 defaultdict 类型; 8. periodicity_data 应初始化为 defaultdict 类型	
test_learning_completion_check	验证 BaselineEngine 能够正确判断学习阶段是否完成， 包括时间条件和样本数量条件的综合评估。	1. 当学习时间超过设定阈值(0.1 秒)时，应满足时间条件; 2. 当样本数量达到最小要求(2 个)时，应满足样本条件; 3. is_learning_complete()方法应返回 True; 4. 学习完成检查应综合考虑时间和样本两个条件; 5. 学习状态判断应准确可靠; 6. 完成条件应符合配置的学习参数	测试通过
test_learning_completion_insufficient_samples	验证 BaselineEngine 在学习时间充足但样本数量不足时， 能够正确判断学习尚未完成。	1. 即使学习时间超过要求(2 秒 > 30 秒配置); 2. 但样本数量只有 50 个(少于要求的 100 个); 3. is_learning_complete()方法应返回 False; 4. 样本数量条件应作为学习完成的必要条件; 5. 学习状态判断应严格遵循样本数量要求; 6. 不应因时间充足而忽略样本数量条件	测试通过
test_learning_completion_insufficient_time	验证 BaselineEngine 在样本数量充足但学习时间不足时， 能够正确判断学习尚未完成。	1. 即使有 150 个样本(超过最小要求 100 个); 2. 但学习时间只有 10 秒(少于要求的 30 秒); 3. is_learning_complete()方法应返回 False; 4. 时间条件应作为学习完成的必要条件; 5. 学习状态判断应严格遵循时间要求; 6. 不应因样本充足而忽略时间条件	测试通过
test_multiple_frames_processing	验证 BaselineEngine 能够正确处理多个不同 CAN ID 的帧， 包括	1. ID 0x123 应记录 2 个时间戳; 2. ID 0x123 的 frame_count 应为	测试通过

	数据分类、统计更新和状态管理等功能。	2; 3. ID 0x123 应记录 DLC 为 8; 4. ID 0x456 应记录 1 个时间戳; 5. ID 0x456 的 frame_count 应为 1; 6. ID 0x456 应记录 DLC 为 4; 7. data_per_id 应包含 2 个不同的 CAN ID; 8. 每个 ID 的数据应正确分离和统计	
test_payload_entropy_calculation	验证 BaselineEngine 能够正确分析 CAN 帧载荷的字节变化模式，包括记录每个字节位置的不同值和计算载荷变化特征。	1. 第 8 个字节(索引 7)应记录 3 个不同的值(0x08, 0x09, 0x0A); 2. 前 7 个字节应保持相同，每个位置只有 1 个唯一值; 3. bytes_at_pos 数据结构应正确记录每个位置的字节值; 4. 载荷变化模式应被正确识别和分析; 5. 字节位置统计应准确反映数据变化特征; 6. 载荷熵计算基础数据应正确收集	测试通过
test_periodicity_detection	验证 BaselineEngine 能够正确检测和记录 CAN 帧的周期性模式，包括时间戳记录和间隔计算等功能。	1. periodicity_data 应正确记录"0x123"的时间戳; 2. 应记录 20 个时间戳，对应 20 个帧; 3. 时间戳间隔应接近设定的周期(0.1 秒); 4. 平均间隔应在 0.1 秒±0.01 秒范围内; 5. 周期性模式应被正确识别和记录; 6. 时间戳序列应保持正确的时间顺序; 7. 周期性数据应为后续分析提供准确基础	测试通过
test_process_frame_for_learning	验证 BaselineEngine 能够正确处理单个 CAN 帧，包括自动启动学习、记录帧数据、更新统计信息等核心学习功能。	1. 处理帧时应自动启动学习阶段(is_learning_active=True); 2. 帧的时间戳应正确记录到 timestamps 列表中; 3. 帧的 DLC 应正确记录到 dlcs 集合中; 4. frame_count 应正确增加到 1; 5. first_seen 时间戳应正确设置; 6. last_seen 时间戳应正确设置; 7. 数据应按 CAN	测试通过

		ID 正确分类存储	
test_reset_learning	验证 BaselineEngine 能够正确重置所有学习相关的状态和数据，将系统恢复到初始状态以便重新开始学习。	1. is_learning_active 应重置为 False; 2. learning_completed 应重置为 False; 3. learning_start_time 应重置为 None; 4. data_per_id 字典应清空，长度为 0; 5. periodicity_data 字典应清空，长度为 0; 6. 所有学习数据应被完全清除; 7. 系统应恢复到可重新开始学习的状态	测试通过
test_start_learning	验证 BaselineEngine 能够正确启动学习阶段，包括设置学习状态标志、记录学习开始时间等关键操作。	1. is_learning_active 应设置为 True; 2. learning_completed 应保持为 False; 3. learning_start_time 应设置为当前时间戳; 4. 学习开始时间应与当前时间相差不超过 1 秒; 5. 学习状态应正确转换为活跃状态; 6. 系统应准备好接收和处理学习数据	测试通过

测试类名：TestConfigManager

测试方法名	描述	预期结果	实际结果
test_add_config_observer	验证 ConfigManager 的观察者模式功能，能够正确添加配置变更观察者。	1. 观察者成功添加到观察者列表	测试通过
test_config_change_notification	验证当配置发生变更时，ConfigManager 能够正确通知所有注册的观察者。	1. 观察者的回调方法被正确调用; 2. 回调参数包含正确的变更信息	测试通过
test_get_all_known_ids	验证 ConfigManager 能够返回所有已知 CAN ID 的集合。	1. 返回 set 类型的已知 ID 集合; 2. 集合包含配置文件中的所有 ID; 3. 集合大小正确	测试通过
test_get_config_version	验证 ConfigManager 的配置版本管理功能，确保版本号正确跟踪配置变更。	1. 初始配置版本为 0; 2. 每次配置修改后版本号正确递增; 3. 版本号跟踪准确	测试通过

test_get_global_setting_no_default	验证当请求的配置设置不存在且未提供默认值时，ConfigManager 抛出 ConfigError 异常。	1. 抛出 ConfigError 异常	测试通过
test_get_global_setting_success	验证 ConfigManager 能够正确获取存在的全局配置设置，包括简单设置和嵌套设置。	测试应该通过	测试通过
test_get_global_setting_with_default	验证当请求的配置设置不存在时，ConfigManager 能够返回指定的默认值。	1. 获取不存在节的设置时返回指定的默认值; 2. 获取存在节中不存在键时返回指定的默认值	测试通过
test_get_id_config_not_found	验证当请求不存在的 CAN ID 配置时的处理行为。	1. 获取不存在的 ID 配置时返回 None	测试通过
test_get_id_config_success	验证 ConfigManager 能够正确获取存在的 CAN ID 的完整配置信息。	1. 成功获取 ID 配置字典; 2. 配置包含正确的名称、DLC 和周期信息	测试通过
test_get_id_setting_success	验证 ConfigManager 能够正确获取特定 CAN ID 的配置设置。	1. 成功获取 0x123 的 expected_dlc 值为 8; 2. 成功获取 0x456 的 name 值为 'Vehicle_Speed'	测试通过
test_get_id_setting_with_default	验证当请求的 ID 配置不存在时，能够使用默认值进行处理。	1. 获取不存在 ID 的配置时返回默认值; 2. 获取存在 ID 的不存在键时返回默认值	测试通过
test_initialization_file_not_found	验证当指定的配置文件不存在时，ConfigManager 能够抛出适当的 ConfigError 异常。	1. 抛出 ConfigError 异常; 2. 异常消息包含 "Configuration file not found" 文本	测试通过
test_initialization_invalid_json	验证当配置文件包含无效的 JSON 格式时，ConfigManager 能够抛出适当的 ConfigError 异常。	1. 抛出 ConfigError 异常; 2. 异常消息包含 "Invalid JSON" 文本	测试通过
test_initialization_success	验证 ConfigManager 能够正确加载有效的配置文件，并初始化所有必要的属性和缓存。	1. ConfigManager 实例创建成功; 2. 文件路径属性设置正确; 3. 配置数据包含所有必要的节; 4. 已知 ID 缓存包含配置文件中的所有 ID; 5. 配置版本号初始化为 0	测试通过
test_is_known_id	验证 ConfigManager 能够正确识别已知和未知的 CAN ID。	1. 已知 ID 返回 True; 2. 未知 ID 返回 False	测试通过
test_reload_config	验证 ConfigManager 能够重新加	1. 内存中的配置修改成功; 2. 重新	测试通过

	载配置文件，丢弃内存中的修改。	加载后配置恢复为文件中的原始值	
test_remove_config_observer	验证 ConfigManager 能够正确移除已添加的配置变更观察者。	1. 观察者成功从观察者列表中移除	测试通过
test_save_config	验证 ConfigManager 能够将修改后的配置正确保存到文件。	1. 配置文件保存成功; 2. 保存的配置内容正确; 3. 新的 ConfigManager 实例能够正确加载保存的配置	测试通过
test_set_global_setting	验证 ConfigManager 能够正确设置全局配置值。	1. 全局配置值设置成功; 2. 能够正确获取设置的值	测试通过
test_set_id_setting	验证 ConfigManager 能够正确设置特定 CAN ID 的配置值。	1. ID 特定配置设置成功; 2. 能够正确获取设置的值	测试通过
test_set_id_setting_new_id	验证 ConfigManager 能够为新的 CAN ID 创建配置并设置值。	1. 新 ID 配置创建成功; 2. 配置值设置正确; 3. ID 被添加到已知 ID 集合	测试通过
test_thread_safety	验证 ConfigManager 在多线程环境下的安全性，确保并发访问不会导致数据竞争或异常。	1. 没有线程安全相关的异常; 2. 所有线程的操作都正确完成; 3. 总操作数符合预期	测试通过
test_update_global_setting	验证 ConfigManager 能够正确更新现有的全局配置设置。	1. 配置值成功更新为新值; 2. 新值与原始值不同	测试通过
test_update_global_setting_new_section	验证 ConfigManager 能够在新创建的配置节中设置配置值。	1. 新配置节创建成功; 2. 新配置值设置成功; 3. 能够正确获取新设置的值	测试通过
test_validation_errors_handling	验证 ConfigManager 能够正确处理包含无效配置值的文件，记录验证错误。	1. ConfigManager 实例创建成功; 2. 验证错误列表包含检测到的错误	测试通过

测试类名: TestDetectionIntegration

测试方法名	描述	预期结果	实际结果
test_alert_severity_distribution	验证在混合攻击场景中，告警的严重性分布是否合理。	1. 应该产生告警; 2. 至少有一些中等或高严重性告警; 3. 严重性分布反映攻击的实际威胁程度	测试通过
test_detector_error_h	验证检测器能够优雅地处理异常	1. 检测器能够处理问题帧而不崩	测试通过

andling	或问题帧，不会崩溃或抛出未处理的异常。	溃; 2. 返回值始终是列表类型; 3. 错误处理机制正常工作	
test_detector_performance_comparison	验证所有检测器在处理大量帧时的性能表现，确保满足实时检测要求。	1. 所有检测器处理时间少于 10 秒; 2. 处理速度至少 50 帧/秒; 3. 性能满足实时检测要求	测试通过
test_drop_attack_detection	验证丢包检测器能够在集成环境中正确检测丢包攻击模式。	1. 应该检测到丢包攻击并产生告警; 2. 告警类型包含 iat_anomaly 等丢包相关类型; 3. 告警数量大于 0	测试通过
test_memory_usage_stability	验证检测系统在处理大量帧时内存使用是否稳定，不会出现内存泄漏。	1. 内存对象增长在合理范围内（<10000 个对象）; 2. 没有明显的内存泄漏; 3. 系统能够长时间稳定运行	测试通过
test_multi_attack_scenario	验证检测系统能够在包含多种攻击类型的复杂场景中正确工作。	1. 应该检测到多种类型的攻击; 2. 至少检测到一种攻击类型; 3. 告警总数大于 0	测试通过
test_normal_traffic_processing	验证所有检测器在处理正常 CAN 流量时的协同工作，确保不产生误报告警。	1. 正常流量产生的告警数量应该很少（≤5 个）; 2. 不应该有高严重性的误报告警; 3. 所有检测器都能正常处理正常流量	测试通过
test_replay_attack_detection	验证重放检测器能够在集成环境中正确检测重放攻击模式。	1. 应该检测到重放攻击并产生告警; 2. 告警类型包含 replay 相关类型; 3. 告警数量大于 0	测试通过
test_state_consistency_across_detectors	验证多个检测器使用相同状态对象时不会相互干扰，保持状态一致性。	1. 状态的基本字段（时间戳、CAN ID）保持一致; 2. 检测器之间不会相互干扰状态; 3. 状态管理正确	测试通过
test_tamper_attack_detection	验证篡改检测器能够在集成环境中正确检测数据篡改攻击。	1. 应该检测到篡改攻击并产生告警; 2. 告警类型包含 tamper 相关类型; 3. 告警数量大于 0	测试通过
test_unknown_id_detection	验证通用规则检测器能够在集成环境中正确检测未知的 CAN ID。	1. 应该检测到未知 ID 并产生告警; 2. 告警类型为 unknown_id_detected; 3. 告警数量大于 0	测试通过

测试类名：TestDropDetector

测试方法名	描述	预期结果	实际结果
test_calculate_current_iat	验证 DropDetector 能够正确计算连续帧之间的时间间隔。	1. 第一帧的 IAT 为 None（没有前一帧）；2. 第二帧的 IAT 等于时间戳差值；3. IAT 计算精度正确	测试通过
test_calculate_iat_z_score	验证 DropDetector 能够正确计算 IAT 的 Z 分数，用于统计异常检测。	1. 正常值的 Z 分数为 0；2. 异常值的 Z 分数反映偏差程度；3. 标准差为 0 时返回无穷大；4. Z 分数计算公式正确	测试通过
test_check_consecutive_missing	验证 DropDetector 能够检测连续丢失的帧数量，并在超过阈值时产生告警。	1. 连续丢失帧数超过阈值时产生告警；2. 告警类型为 consecutive_missing_frames；3. 告警包含丢失帧数信息	测试通过
test_check_dlc_zero_special	验证 DropDetector 对 DLC 为 0 的帧进行特殊处理，检测其时序异常。	1. DLC 为 0 且 IAT 异常时产生告警；2. 告警类型为 dlc_zero_timing_anomaly；3. 特殊处理逻辑正确执行	测试通过
test_check_iat_anomaly_abnormal	验证当 IAT 超出正常范围时，DropDetector 能够正确检测并产生相应严重级别的告警。	1. 异常 IAT 产生 iat_anomaly 类型告警；2. 告警严重级别根据偏差程度确定；3. 大幅偏差产生 HIGH 级别告警	测试通过
test_check_iat_anomaly_normal	验证当 IAT 在正常范围内时，DropDetector 不会产生异常告警。	1. 正常 IAT 不产生任何告警；2. 告警列表为空	测试通过
test_check_max_iat_factor	验证 DropDetector 能够检测 IAT 相对于平均值的倍数，并在超过最大因子时产生告警。	1. IAT 超过平均值的最大倍数时产生告警；2. 告警类型为 iat_max_factor_violation；3. 告警包含因子信息	测试通过
test_detect_disabled	验证当 drop 检测在配置中被禁用时，检测器不执行任何检测逻辑。	1. 检测被禁用时不产生任何告警；2. 检测器正确响应配置变更；3. 性能优化（跳过检测逻辑）	测试通过
test_detect_drop_attack_sequence	验证 DropDetector 能够检测模拟的丢包攻击序列，识别异常的帧间间隔模式。	1. 检测到丢包攻击并产生告警；2. 告警类型包含 iat_anomaly；3. 攻击模式被正确识别	测试通过
test_detect_no_learne	验证当 CAN ID 没有学习数据	1. 没有学习数据时不产生告警；2.	测试通过

d_data	时，DropDetector 的检测行为是否正确。	检测器优雅处理未知 ID; 3. 不抛出异常	
test_detector_initialization	验证 DropDetector 实例能够正确初始化，设置正确的检测器类型和初始状态。	1. 检测器类型为 'drop'; 2. IAT 异常计数初始化为 0; 3. 连续丢失计数初始化为 0; 4. 最大 IAT 违规计数初始化为 0	测试通过
test_estimate_missing_frames	验证 DropDetector 能够根据 IAT 和学习统计数据准确估算丢失的帧数。	1. 正常 IAT 估算丢失帧数为 0; 2. 异常 IAT 估算丢失帧数大于 0; 3. 更大的 IAT 估算更多的丢失帧; 4. 估算算法逻辑正确	测试通过
test_get_learned_iat_stats	验证 DropDetector 能够正确获取已知 CAN ID 的学习 IAT 统计数据，并正确处理未知 ID。	1. 已知 ID 返回包含 mean_iat 和 std_iat 的字典; 2. 统计数据值与配置文件中的值匹配; 3. 未知 ID 返回 None	测试通过
test_performance_with_large_sequence	验证 DropDetector 在处理大量帧序列时的性能表现，确保检测效率。	1. 1000 帧处理时间少于 5 秒; 2. 检测器性能满足实时要求; 3. 正常帧不产生误报告警; 4. 内存使用稳定	测试通过
test_update_drop_state	验证 DropDetector 能够正确更新检测状态，包括检测时间和计数。	1. 检测时间字段被正确设置; 2. 检测计数字段被正确设置; 3. 状态更新逻辑正确	测试通过

测试类名：TestGeneralRulesDetector

测试方法名	描述	预期结果	实际结果
test_auto_add_id_to_baseline	验证 GeneralRulesDetector 能够正确执行将 ID 自动添加到基线的操作。	1. 调用基线引擎的自动添加方法; 2. Shadow 学习状态标记为已添加; 3. 自动添加计数增加	测试通过
test_auto_add_threshold	验证当 Shadow 学习的帧数达到阈值时，系统能够自动将 ID 添加到基线。	1. 达到阈值时调用自动添加方法; 2. Shadow 学习状态标记为已添加到基线; 3. 基线引擎的相关方法被正确调用	测试通过
test_check_unknown_id_disabled	验证当未知 ID 检测在配置中被禁用时，检测器不执行检测逻辑。	1. 检测被禁用时不产生任何告警; 2. 检测器正确响应配置变更	测试通过

test_check_unknown_id_known	验证当处理已知 CAN ID 的帧时，GeneralRulesDetector 不会产生未知 ID 告警。	1. 已知 ID 不产生任何告警; 2. 告警列表为空	测试通过
test_check_unknown_id_unknown	验证当处理未知 CAN ID 的帧时，GeneralRulesDetector 能够正确检测并产生告警。	1. 未知 ID 产生 unknown_id_detected 类型告警; 2. 告警严重级别为 HIGH; 3. 告警数量为 1	测试通过
test_detect_disabled	验证当通用规则检测在配置中被完全禁用时，检测器不执行任何检测逻辑。	1. 检测被禁用时不产生任何告警; 2. 检测器正确响应配置变更; 3. 性能优化（跳过检测逻辑）	测试通过
test_detect_unknown_id_sequence	验证 GeneralRulesDetector 能够正确处理连续的未知 ID 帧序列。	1. 检测到未知 ID 并产生告警; 2. 告警类型为 unknown_id_detected; 3. 告警数量大于 0	测试通过
test_detector_initialization	验证 GeneralRulesDetector 实例能够正确初始化，设置正确的检测器类型和初始状态。	1. 检测器类型为 'general_rules'; 2. 各种计数器初始化为 0; 3. 基线引擎正确关联; 4. Shadow 学习状态初始化为空字典	测试通过
test_get_unknown_id_settings	验证 GeneralRulesDetector 能够正确从配置管理器中获取未知 ID 检测的相关设置。	1. 返回字典类型的设置; 2. 包含 learning_mode 等关键配置项; 3. 配置值与预期一致	测试通过
test_post_detect	验证 GeneralRulesDetector 在执行检测后的后处理步骤是否正确更新状态。	1. ID 状态中添加检测时间字段; 2. ID 状态中添加检测计数字段; 3. 状态更新逻辑正确	测试通过
test_pre_detect	验证 GeneralRulesDetector 在执行检测前的预处理步骤是否正确。	1. 返回 True 表示继续检测; 2. 检测计数正确更新; 3. 最后检测时间被设置	测试通过
test_shadow_learning_mode	验证 GeneralRulesDetector 在 Shadow 学习模式下能够正确学习未知 ID 并更新状态。	1. 产生告警的同时添加到 Shadow 学习状态; 2. Shadow 学习状态正确记录帧计数; 3. 调用基线引擎的 Shadow 学习方法	测试通过
test_should_auto_add_id	验证 GeneralRulesDetector 能够正确判断何时应该将未知 ID 自动添加到基线。	1. 满足条件时返回 True; 2. 已添加过的 ID 返回 False; 3. 判断逻辑正确	测试通过

test_update_shadow_learning_state	验证 GeneralRulesDetector 能够正确维护和更新 Shadow 学习状态。	1. 首次更新正确初始化状态; 2. 帧计数从 1 开始; 3. 添加到基线标志初始为 False; 4. 后续更新正确增加计数	测试通过

测试类名: TestReplayDetector

测试方法名	描述	预期结果	实际结果
test_calculate_payload_repetition_score	测试计算载荷重复分数功能	1. 重复次数多的载荷应有更高的分数; 2. 分数应反映载荷的重复程度	测试通过
test_check_contextual_payload_repetition_abnormal	验证 ReplayDetector 对载荷重复攻击的检测能力，确保能识别异常的载荷重复模式。	1. 应产生 1 个载荷重复告警; 2. 告警类型应为 "replay_identical_payload"	测试通过
test_check_contextual_payload_repetition_normal	验证 ReplayDetector 对正常载荷变化的检测行为，确保不同载荷不产生误报。	1. 首次出现的载荷不应产生告警; 2. 返回的告警列表应为空	测试通过
test_check_fast_replay_enhanced_no_periodicity	测试增强快速重放检测功能（无周期性数据场景）	1. 应返回列表类型的结果; 2. 当无周期性数据时应回退到传统检测	测试通过
test_check_fast_replay_legacy	测试传统快速重放检测功能	1. 应产生 1 个快速重放告警; 2. 告警类型应为 "replay_fast_replay"	测试通过
test_check_sequence_replay_abnormal	验证 ReplayDetector 对序列重放攻击的检测能力，确保能识别重复的帧序列模式。	1. 重复序列应产生告警; 2. 应包含序列重放类型的告警	测试通过
test_check_sequence_replay_normal	验证 ReplayDetector 对正常序列的检测行为，确保不重复的序列不产生误报。	1. 正常序列不应产生任何告警; 2. 总告警数量应为 0	测试通过
test_detect_disabled	验证 ReplayDetector 在被禁用时的行为，确保配置控制功能正常工作。	1. 当检测器被禁用时不应产生任何告警; 2. 返回的告警列表应为空	测试通过
test_detect_replay_attack_sequence	验证 ReplayDetector 对重放攻击序列的检测能力，确保能识别混合在正常流量中的重放攻击。	1. 应检测到重放攻击并产生告警; 2. 告警类型应包含"replay"相关类型	测试通过

test_detector_initialization	验证 ReplayDetector 检测器的初始化过程，确保所有属性和状态正确设置。	1. detector_type 应为'replay'; 2. 所有攻击计数器应为 0; 3. _periodicity_cache 应为 dict 类型	测试通过
test_find_sequence_patterns	测试查找序列模式功能	1. 应找到至少一个重复模式; 2. 应正确识别 ABC 重复模式	测试通过
test_get_periodicity_baseline	测试获取周期性基线数据功能	1. 已知 ID 应返回有效的周期性数据; 2. 未知 ID 应返回 None	测试通过
test_is_periodic_pattern	测试周期性模式判断功能	1. 接近学习均值的 IAT 应被判断为周期性; 2. 远离学习均值的 IAT 应被判断为非周期性	测试通过
test_performance_with_large_sequence	验证 ReplayDetector 在处理大量帧序列时的性能表现，确保检测效率满足实时要求。	1. 处理时间应在合理范围内; 2. 检测器应能处理大量帧而不出错; 3. 内存使用应保持稳定	测试通过
test_periodicity_cache_cleanup	测试周期性缓存清理	测试应该通过	测试通过
test_update_payload_history	测试更新载荷历史功能	1. 首次添加的哈希计数应为 1; 2. 相同哈希再次添加时计数应递增; 3. 不同哈希应创建新的历史条目	测试通过
test_update_replay_state	验证 ReplayDetector 状态更新机制的正确性，确保检测状态信息得到正确维护。	1. 应添加重放检测相关的状态字段; 2. 载荷哈希值应正确保存	测试通过
test_update_sequence_buffer	测试更新序列缓冲区功能	1. 缓冲区长度不应超过最大长度限制; 2. 应保留最新添加的元素	测试通过

测试类名: TestStateManager

测试方法名	描述	预期结果	实际结果
test_calculate_iat	验证 StateManager 能够正确计算连续 CAN 帧之间的时间间隔，第一帧不应有 IAT 值，后续帧应正确计算与前一帧的时间差。	1. 第一帧处理后状态中不应包含 last_iat 字段; 2. 第二帧处理后应正确计算 IAT 值; 3. IAT 值应等于两帧时间戳的差值(0.15 秒); 4. IAT 计算应精确到毫秒级别; 5. 计算过程不应抛出异常; 6. IAT 值应为正数	测试通过

test_cleanup_inactive_ids	验证 StateManager 的内部清理机制能够正确识别和移除不活跃的 ID 状态， 基于活跃时间阈值进行智能清理，优化内存使用。	1. 不活跃的 ID 状态应被正确识别; 2. 清理操作应成功移除不活跃的 ID; 3. 活跃的 ID 状态应保持不变; 4. 清理后总 ID 数量应减少; 5. 清理逻辑应基于 last_active 时间戳; 6. 清理过程应保持数据一致性	测试通过
test_cleanup_old_data	验证 StateManager 能够正确识别和清理长时间未活跃的 ID 状态， 保留活跃的 ID 状态，确保内存使用效率和系统性能。	1. 长时间未活跃的 ID 状态应被正确识别; 2. 旧的 ID 状态应被成功清理; 3. 活跃的 ID 状态应被保留; 4. 清理后 ID 总数应减少; 5. 清理操作应基于 last_active 时间戳; 6. 清理过程不应影响活跃状态的完整性	测试通过
test_clear_all_states	验证 StateManager 能够正确清空所有 ID 状态信息， 将系统重置到初始状态，释放所有占用的内存。	1. 清空前应确认存在多个 ID 状态; 2. 清空操作应成功执行; 3. 清空后 id_states 字典应为空; 4. 所有 ID 状态数据应被完全清除; 5. 系统应恢复到初始状态; 6. 清空操作不应抛出异常	测试通过
test_force_remove_oldest_id	验证 StateManager 在内存压力下能够强制移除最旧的 ID 状态，基于 first_seen 时间戳确定最旧的 ID 并安全移除。	1. 应成功移除一个 ID 状态; 2. 移除的应该是 first_seen 时间最早的 ID; 3. 总 ID 数量应减少 1; 4. 最旧的 ID(id_0)应从状态管理器中移除; 5. 其他 ID 状态应保持完整; 6. 强制移除操作不应抛出异常	测试通过
test_get_id_state	验证 StateManager 能够正确获取指定 ID 的状态信息， 对存在的 ID 返回完整状态，对不存在的 ID 返回 None。	1. 存在的 ID 应返回非空状态对象; 2. 返回的状态应包含正确的 frame_count 值; 3. 不存在的 ID 应返回 None; 4. 获取操作不应修改状态数据; 5. 方法应正确处理各种 ID 格式; 6. 获取过程不应抛出异常	测试通过
test_get_stats	验证 StateManager 能够正确收集和返回系统运行统计信息， 包括	1. 统计信息应正确反映实际操作次数; 2. total_updates 应等于实	测试通过

	更新次数、活跃 ID 数量、清理次数等关键指标。	际更新次数(5); 3. active_ids 应等于当前活跃 ID 数量(5); 4. 统计信息应包含所有必要字段; 5. 返回的统计数据应为字典格式; 6. 统计信息获取不应抛出异常	
test_initialize_id_state	验证 StateManager 能够正确为新的 CAN ID 创建和初始化状态记录， 包括设置首次见到时间、帧计数、时间戳和各种历史记录缓冲区。	1. 新 ID 应成功添加到 id_states 字典中; 2. first_seen 时间戳应正确设置; 3. frame_count 应初始化为 0; 4. last_timestamp 应设置为提供的时间戳; 5. last_active 时间应正确设置; 6. payload_hashes 列表应初始化为空列表; 7. sequence_buffer 列表应初始化为空列表; 8. 所有数据结构类型应符合预期	测试通过
test_limit_payload_hashes	验证 StateManager 能够正确限制每个 ID 状态中存储的载荷哈希数量， 防止内存无限增长， 保持系统性能稳定。	1. 载荷哈希列表长度应被限制在最大值以内; 2. 超出限制的哈希应被移除; 3. 限制操作应保留最新的哈希值; 4. 限制过程不应影响其他状态数据; 5. 内存使用应得到有效控制; 6. 限制操作不应抛出异常	测试通过
test_limit_sequence_buffer	验证 StateManager 能够正确限制每个 ID 状态中序列缓冲区的大小， 防止缓冲区无限增长， 确保内存使用效率。	1. 序列缓冲区长度应被限制在最大值以内; 2. 超出限制的序列数据应被移除; 3. 限制操作应保留最新的序列数据; 4. 限制过程不应影响其他状态信息; 5. 缓冲区管理应高效执行; 6. 限制操作不应抛出异常	测试通过
test_max_ids_limit	验证 StateManager 能够正确执行最大 ID 数量限制， 当 ID 数量超过设定阈值时， 应自动清理旧的 ID 状态以保持内存使用在合理范围内。	1. 创建超过 max_ids 限制的 ID 时应触发清理机制; 2. 最终 ID 数量不应超过设定的最大值(3 个); 3. 清理过程应优先移除最旧的 ID 状态; 4. 清理操作不应影响系统稳定性; 5. 内存管理应正确执行; 6. 清理过程不应抛出异常	测试通过

test_performance_with_large_dataset	验证 StateManager 在处理大量 CAN 帧数据时的性能表现， 确保系统能够在合理时间内处理大规模数据集。	1. 1000 帧数据应在 5 秒内完成处理; 2. 最终应正确管理 50 个不同的 ID 状态; 3. 统计信息应正确反映 1000 次更新; 4. 处理过程应保持高效率; 5. 大数据集处理不应导致内存泄漏; 6. 性能测试不应抛出异常	测试通过
test_periodic_cleanup	验证 StateManager 的定期清理机制能够按照设定的时间间隔自动触发， 在系统运行过程中自动维护内存使用效率。	1. 清理间隔应正确设置为 1 秒; 2. 超过清理间隔后的更新操作应触发清理; 3. last_cleanup_time 应正确更新; 4. 清理机制应自动运行; 5. 定期清理不应影响正常操作; 6. 清理过程应保持系统稳定性	测试通过
test_remove_id_state	验证 StateManager 能够正确移除指定 ID 的状态信息， 成功移除存在的 ID 并正确处理不存在的 ID 移除请求。	1. 存在的 ID 应成功移除并返回 True; 2. 移除后 ID 不应存在于状态管理器中; 3. 不存在的 ID 移除请求应返回 False; 4. 移除操作不应影响其他 ID 状态; 5. 移除过程应完全清理相关数据; 6. 移除操作不应抛出异常	测试通过
test_state_manager_initialization	验证 StateManager 类能够正确初始化， 包括设置最大 ID 数量、清理间隔、 初始化内部数据结构和统计信息等关键参数。	1. max_ids 参数应正确设置为 100; 2. cleanup_interval 参数应正确设置为 60 秒; 3. id_states 字典应正确初始化为空字典; 4. 统计信息应正确初始化， 所有计数器为 0; 5. 内部数据结构类型应符合预期; 6. 初始化过程不应抛出异常	测试通过
test_update_and_get_state_existing_id	验证 StateManager 在处理已存在 CAN ID 的后续帧时， 能够正确更新现有状态记录， 包括计算帧间间隔(IAT)、更新帧计数和时间戳等信息。	1. 应返回与第一次相同的状态对象引用; 2. last_timestamp 应更新为最新帧的时间戳; 3. frame_count 应正确递增到 2; 4. last_iat 应正确计算为两帧间的时间差; 5. 统计信息中 total_updates 应增加到 2; 6. 状态更新过程应保	测试通过

		持数据一致性	
test_update_and_get_state_new_id	验证 StateManager 在处理新 CAN ID 的第一个帧时，能够正确创建状态记录并返回初始化后的状态信息，同时更新相关统计数据。	1. 方法应成功返回非空状态对象; 2. last_timestamp 应设置为帧的时间戳; 3. frame_count 应更新为 1; 4. 统计信息中 total_updates 应增加 1; 5. 新 ID 状态应正确存储在内部数据结构中; 6. 状态更新过程不应抛出异常	测试通过

测试类名: TestSystemEndToEnd

测试方法名	描述	预期结果	实际结果
test_alert_output	验证系统能够正确生成和输出警报信息，包括警报生成、警报管理、文件输出和 JSON 格式验证等功能。	1. 应成功处理帧数据并可能生成警报; 2. 警报应正确发送到警报管理器; 3. 警报刷新操作应成功执行; 4. 警报文件应正确生成(如果有警报); 5. 警报 JSON 格式应正确; 6. 警报数据应包含必要字段(timestamp, alert_type, severity); 7. 警报输出应保持数据完整性; 8. 警报处理不应抛出异常	测试通过
test_config_loading	验证系统能够正确加载配置文件，包括配置管理器初始化、配置版本验证、各检测器配置获取和 ID 特定设置访问等功能。	1. 配置管理器应成功初始化且非空; 2. 配置版本应成功获取且非空; 3. drop 检测器配置应正确获取; 4. tamper 检测器配置应正确获取; 5. replay 检测器配置应正确获取; 6. 各检测器启用状态应为布尔类型; 7. 已知 ID 的特定配置应正确获取; 8. 配置加载过程不应抛出异常	测试通过
test_data_file_parsing	验证系统能够正确解析 CAN 数据文件，包括逐行读取、帧结构验证、数据完整性检查等关键解析功能。	1. 应成功读取到数据行(至少 1 行); 2. 应解析出有效的 CAN 帧(至少 1 个); 3. 解析的帧应为 CANFrame 类型; 4. 帧的	测试通过

		timestamp 字段应非空; 5. 帧的 can_id 字段应非空; 6. 帧的 dlc 字段应非空; 7. 帧的 payload 字段应非空; 8. 解析过程不应抛出异常	
test_detection_pipeline	验证完整的 CAN 帧检测流水线，包括帧解析、状态更新、 多检测器运行、警报生成和警报管理等端到端流程。	1. 应成功处理多个 CAN 帧(至少 1 个); 2. 状态管理器应正确更新帧状态; 3. 所有检测器应成功运行; 4. 检测过程可能生成警报; 5. 警报应正确发送到警报管理器; 6. 警报类型分布应正确统计; 7. 检测流水线应保持稳定运行; 8. 处理过程不应抛出异常	测试通过
test_memory_usage	验证系统在处理大量 CAN 帧时的内存使用效率， 监控内存增长情况并确保内存使用在合理范围内。	1. 应成功处理大量帧数据(5000 帧); 2. 状态管理器应正确更新所有帧状态; 3. 所有检测器应正常运行; 4. 内存增长应在合理范围内(<100MB); 5. 垃圾回收应正确执行; 6. 每帧平均内存使用应可控; 7. 内存监控应准确记录; 8. 内存测试不应抛出异常	测试通过
test_system_performance	验证系统的整体性能表现， 包括处理速度、错误率、 吞吐量等关键性能指标的测量和评估。	1. 应成功处理大量帧数据(2000 帧); 2. 处理时间应被正确记录; 3. 处理速度应达到合理水平(>1 帧/秒); 4. 解析错误应在可接受范围内; 5. 系统应保持稳定运行; 6. 性能指标应准确计算; 7. 错误处理应正确执行; 8. 性能测试不应抛出异常	测试通过

测试类名: TestSystemPerformance

测试方法名	描述	预期结果	实际结果
test_concurrent_processing	验证系统的并发处理能力， 测试	1. 应成功启动多个并发处理线程	测试通过

	多线程处理、资源竞争和同步机制等并发特性。	(4 个); 2. 每个线程应处理指定数量的帧(1000 帧); 3. 线程间应正确同步和协调; 4. 总处理时间应合理; 5. 并发效率应优于串行处理; 6. 不应出现资源竞争问题; 7. 线程安全应得到保证; 8. 并发测试不应抛出异常	
test_cpu_usage_efficiency	验证系统在处理 CAN 帧时的 CPU 使用效率， 监控 CPU 负载、处理效率和资源利用率等指标。	1. 应成功处理大量帧数据(8000 帧); 2. CPU 使用率应被准确监控; 3. 平均 CPU 使用率应在合理范围内(<90%); 4. 最大 CPU 使用率应可控(<98%); 5. CPU 效率应保持稳定; 6. 处理负载应均匀分布; 7. 系统响应应保持流畅; 8. CPU 效率测试不应抛出异常	测试失败: AssertionError: 100.8 not less than 95 : 最大 CPU 使用率应该小于 95%
test_detector_individual_performance	验证各个检测器的独立性能表现， 分别测试 drop、tamper 和 replay 检测器的处理效率和准确性。	1. 应成功测试所有检测器(drop, tamper, replay); 2. 每个检测器应处理指定数量的帧(2000 帧); 3. 各检测器处理时间应被准确记录; 4. 检测器性能应达到合理水平; 5. 各检测器应独立正常工作; 6. 检测准确性应保持稳定; 7. 性能指标应准确计算; 8. 检测器性能测试不应抛出异常	测试通过
test_latency_performance	验证系统处理 CAN 帧的延迟性能， 测量单帧处理时间、响应延迟和处理一致性等指标。	1. 应成功处理多个帧数据(1000 帧); 2. 每帧处理时间应被准确记录; 3. 平均处理延迟应在合理范围内(<10ms); 4. 最大处理延迟应可控(<50ms); 5. 延迟分布应相对均匀; 6. 处理时间应保持一致性; 7. 系统响应应及时; 8. 延迟测试不应抛出异常	测试通过
test_long_running_stability	验证系统长时间连续运行的稳定性， 监控系统状态、资源使用和	1. 应成功运行指定时间(30 秒); 2. 系统应保持持续稳定运行; 3. 处理	测试通过

	性能衰减等长期运行指标。	性能应保持一致; 4. 内存使用应保持稳定; 5. 不应出现性能衰减; 6. 系统状态应保持正常; 7. 资源使用应保持合理; 8. 长期运行不应导致系统异常	
test_memory_usage_stability	验证系统长时间运行时的内存使用稳定性， 监控内存泄漏、垃圾回收效果和内存增长趋势。	1. 应成功处理大量帧数据(15000 帧); 2. 内存使用应保持相对稳定; 3. 内存增长应在合理范围内 (<200MB); 4. 垃圾回收应有效执行; 5. 不应出现明显的内存泄漏; 6. 每帧平均内存使用应可控; 7. 内存监控应准确记录; 8. 内存稳定性测试不应抛出异常	测试通过
test_stress_performance	验证系统在高负载压力下的性能表现， 测试系统稳定性、错误处理和恢复能力等关键指标。	1. 应成功处理超大量帧数据 (20000 帧); 2. 系统应保持稳定运行; 3. 处理速度应保持在合理水平; 4. 错误率应在可接受范围内; 5. 内存使用应保持可控; 6. CPU 使用应保持合理; 7. 系统应正确处理压力负载; 8. 压力测试不应导致系统崩溃	测试通过
test_system_resource_limits	验证系统在资源限制条件下的表现， 测试内存限制、CPU 限制和处理能力边界等资源约束场景。	1. 应成功处理大量帧数据(25000 帧); 2. 系统应正确处理资源压力; 3. 内存使用应在限制范围内; 4. CPU 使用应保持合理; 5. 系统应优雅处理资源不足; 6. 性能应在可接受范围内; 7. 错误处理应正确执行; 8. 资源限制测试不应导致系统崩溃	测试通过
test_throughput_performance	验证系统在处理大量 CAN 帧时的吞吐量性能， 测量处理速度、帧率和系统响应时间等关键指标。	1. 应成功处理大量帧数据(10000 帧); 2. 处理时间应被准确记录; 3. 平均处理速度应达到合理水平 (>100 帧/秒); 4. 系统应保持稳定的吞吐量; 5. 内存使用应保持稳	测试通过

		定; 6. 处理延迟应在可接受范围内; 7. 性能指标应准确计算; 8. 吞吐量测试不应抛出异常	

测试类名: TestTamperDetector

测试方法名	描述	预期结果	实际结果
test_analyze_byte_behavior	验证 TamperDetector 字节行为分析功能的正确性，确保能准确识别不同类型的字节行为模式。	1. 应返回字节行为分析结果; 2. 结果应包含各字节的行为特征	测试通过
test_calculate_byte_change_ratio	验证 TamperDetector 字节变化率计算算法的准确性，确保能正确量化载荷变化程度。	1. 部分变化载荷的变化率应为 0.5; 2. 相同载荷的变化率应为 0.0; 3. 完全不同载荷的变化率应为 1.0	测试通过
test_check_byte_behavior_counter_byte	验证 TamperDetector 对正常计数器字节行为的检测，确保符合计数器模式的字节不产生误报。	1. 正常的计数器字节行为不应产生告警; 2. 计数器字节相关告警列表应为空	测试通过
test_check_byte_behavior_static_byte	验证 TamperDetector 对符合期望的静态字节的检测行为，确保正常静态字节不产生误报。	1. 符合期望的静态字节不应产生告警; 2. 静态字节相关告警列表应为空	测试通过
test_check_byte_behavior_static_byte_mismatch	验证 TamperDetector 对静态字节篡改的检测能力，确保能识别静态字节值的异常变化。	1. 不符合期望的静态字节应产生告警; 2. 应产生一个静态字节不匹配告警	测试通过
test_check_byte_change_ratio_abnormal	验证 TamperDetector 对异常字节变化率的检测能力，确保能识别大幅载荷篡改。	1. 应产生一个字节变化率异常告警; 2. 告警类型应为 tamper_byte_change_ratio	测试通过
test_check_byte_change_ratio_normal	验证 TamperDetector 对正常字节变化率的检测行为，确保小幅变化不产生误报。	1. 小幅字节变化不应产生告警; 2. 告警列表应为空	测试通过
test_check_dlc_anomaly_abnormal	验证 TamperDetector 对异常 DLC 值的检测能力，确保能正确识别 DLC 篡改攻击。	1. 应产生一个 DLC 异常告警; 2. 告警类型应为 tamper_dlc_anomaly; 3. 告警严重级别应为 HIGH	测试通过

test_check_dlc_anomaly_normal	验证 TamperDetector 对正常 DLC 值的检测行为，确保不产生误报。	1. 正常 DLC 值不应产生任何告警; 2. 告警列表应为空	测试通过
test_check_entropy_anomaly_abnormal	验证 TamperDetector 对高熵载荷的检测能力，确保能识别可能的载荷篡改。	1. 高熵载荷可能产生熵异常告警（取决于具体熵值）; 2. 函数应正常执行不崩溃; 3. 返回值应为告警列表	测试通过
test_check_entropy_anomaly_normal	验证 TamperDetector 对正常熵值载荷的检测行为，确保低熵载荷不产生误报。	1. 低熵载荷不应产生熵异常告警; 2. 告警列表应为空	测试通过
test_detect_disabled	验证 TamperDetector 在被禁用时的行为，确保配置控制功能正常工作。	1. 检测被禁用时不应产生任何告警	测试通过
test_detect_no_learned_data	验证 TamperDetector 在缺少学习数据时的检测行为，确保系统能优雅处理未知 ID。	1. 应返回告警列表（可能为空或包含 DLC 异常）; 2. 没有学习数据时只能进行基本检测	测试通过
test_detect_small_dlc_skip_payload_analysis	验证 TamperDetector 对小 DLC 帧的处理逻辑，确保跳过不必要的载荷分析以提高效率。	1. 可能产生 DLC 相关告警; 2. 不应产生载荷相关告警（熵、字节行为等）	测试通过
test_detect_tamper_attack_sequence	验证 TamperDetector 对篡改攻击序列的检测能力，确保能识别混合在正常流量中的篡改攻击。	1. 应检测到篡改攻击并产生告警; 2. 告警类型应包含篡改相关类型	测试通过
test_detector_initialization	验证 TamperDetector 检测器的初始化过程，确保所有属性和状态正确设置。	1. 检测器类型应为'tamper'; 2. 所有异常计数器应初始化为 0; 3. 基线引擎应正确关联	测试通过
test_performance_with_large_sequence	验证 TamperDetector 在处理大量帧序列时的性能表现，确保检测效率满足实时要求。	1. 处理时间应在 10 秒内完成; 2. 检测器应能稳定处理大量帧; 3. 内存使用应保持合理	测试通过
test_update_tamper_state	验证 TamperDetector 状态更新机制的正确性，确保检测状态信息得到正确维护。	1. 应更新最后检测时间; 2. 应更新检测计数; 3. 应更新最后载荷	测试通过