# ДОДАТОК А

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
МІНІСТЕРСТВО ОСВІТИ І НАУКИ РЕСПУБЛІКИ
КАЗАХСТАН МІНІСТЕРСТВО ОСВІТИ
АЗЕРБАЙДЖАНСЬКОЇ РЕСПУБЛІКИ МІНІСТЕРСТВО
ВИЩОЇ І СЕРЕДНЬОЇ СПЕЦІАЛЬНОЇ ОСВІТИ
РЕСПУБЛІКИ УЗБЕКИСТАН НАЦІОНАЛЬНИЙ
ТЕХНІЧНИЙ УНІВЕРСИТЕТ "ХАРКІВСЬКИЙ
ПОЛІТЕХНІЧНИЙ ІНСТИТУТ" ДОНБАСЬКА ДЕРЖАВНА
МАШИНОБУДІВНА АКАДЕМІЯ**

---

**ІНФОРМАТИКА, УПРАВЛІННЯ ТА ШТУЧНИЙ ІНТЕЛЕКТ**

**ТЕЗИ ВОСЬМОЇ МІЖНАРОДНОЇ НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ**
***(16 – 19 листопада 2021 року)***

Харків – Краматорськ
2021

## ТЕХНОЛОГІЯ КОМБІНОВАНОГО АНАЛІЗУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ЯК ПЕРСПЕКТИВНИЙ НАПРЯМ ПОШУКУ ВРАЗЛИВОСТЕЙ ПІД ЧАС ЙОГО РОЗРОБКИ

*канд. техн. наук, доц. О.І. Федюшин, магістр В.В. Ярещенко, Харківський національний університет радіоелектроніки, м. Харків*

На сьогодні не існує універсальної методики, яка б дозволяла всебічно проводити тестування програмного забезпечення (ПЗ) за мінімальний час. Для підвищення ефективності аналізу пропонується розглянути комбінацію інструментів статичного аналізу вихідного коду[1-3] та динамічного і статичного аналізу бінарного коду [2-4].

Дослідження показало, що засоби статичного аналізу є досить ефективними не тільки на ранніх стадіях розробки ПЗ, основним питанням є проблема відсіювання хибних висновків. Для її вирішення пропонується точкова обробка звітів про проблеми інструментами динамічного аналізу. І навпаки, для вирішення проблеми динамічного аналізу – генерації вхідних даних з метою більш повного покриття коду, пропонується використовувати дані з попередніх етапів статичного аналізу коду.

Додатково для вирішення проблеми невідповідності вихідного коду реальному бінарному коду програми – пропонується використовувати порівняння внутрішніх представлень згенерованих під час аналізу вихідного та об’єктного коду [3].

Результати досліджень свідчать, що комбінація методів аналізу між собою дозволяє мінімізувати недоліки окремих підходів до тестування ПЗ без значних втрат в якості покриття тестовими сценаріями.

**Список літератури: 1.** *Black, P. , Okun, V. and Guttman, B.* (2021), Guidelines on Minimum Standards for Developer Verification of Software, NIST Interagency/Internal Report (NISTIR), National Institute of Standards and Technology, Gaithersburg, MD. https://doi.org/10.6028/NIST.IR.8397. **2.** *Аветисян А. И., Белеванцев А.А., Чекляев И.И.* Технологии статического и динамического анализа программного обеспечения //Вопросы кибербезопасности. –2014. –№3(4). –С. 20-28. **3.** *Федюшин О.І., Рибкін І.С.* Методи виявлення вразливостей програмного забезпечення пристроїв IoT від загроз несанкціонованого доступу/ Інформатика, управління та штучний інтелект. Матеріали п’ятої міжнародної науково-технічної конференції студентів, магістрів та аспірантів.– Харків: НТУ "ХПІ", 2018. –С.92.

# ДОДАТОК Б

## Лістинг програмного коду

```
class MainWindow : public QMainWindow
{
Q_OBJECT
public:
    MainWindow(QWidget* parent = nullptr);
private:
    void setupUI();
    QWidget* createSourcesAnalysisWidgetsGroup();
    QWidget* createBinariesAnalysisWidgetsGroup();
    QWidget* createRunAsyncAnalysisPredicateWidget();
    QWidget* createOutputWidgetsBlock();
    QWidget* createAnalysisRunningStateWidget();

    OperationsPool* performAnalysis(bool runAsync);
private:
    AnalysisCaller _caller;
    QCheckBox* _enableStaticSourcesCheckbox = nullptr;
    QLineEdit* _pathToSources = nullptr;
    QCheckBox* _enableStaticBinaryCheckbox = nullptr;
    QCheckBox* _enableDynamicBinaryCheckbox = nullptr;
    QLineEdit* _pathToBinary = nullptr;
    QCheckBox* _runAsyncCheckbox = nullptr;
    QTextBrowser* _sourcesAnalysisOutputBrowser = nullptr;
    QTextBrowser* _binaryAnalysisOutputBrowser = nullptr;
    QLabel* _binaryErrorsSummary = nullptr;
    QLabel* _timeTakenLabel = nullptr;
```

```
    QProgressBar* _progressBar = nullptr;
};

namespace
{
class AsyncOperation : public QObject
{
    Q_OBJECT
public:
    AsyncOperation(std::function<QVariant()> operation, const bool runAsync = true)
    : _operation(std::move(operation))
    , _runAsync(runAsync)
    {
    }

    void run()
    {
        if (_runAsync)
        {
            QThread *thread = new QThread();
            moveToThread(thread);

            connect(thread, &QThread::started, this, [this, thread] {
                // execute
                _result = _operation();
                thread->quit();
                // inform
                emit finished(_result);
                // cleanup
                thread->deleteLater();
```

```
            deleteLater();
        });


        thread->start();
    }
    else
    {
        _result = _operation();

        emit finished(_result);

        deleteLater();
    }
}

signals:
    void finished(QVariant);

private:
    bool _runAsync = true;
    QVariant _result;
    std::function<QVariant()> _operation;
};

void extractAnalysisErrorsSummary(const QString& analysisOutput, QLabel*
destination)
{
    // Dynamic output example: ERROR SUMMARY: 0 errors from 0 contexts
(suppressed: 0 from 0)
```

```
    auto dynamicCount = -1;

    static const QRegularExpression DynamicErrorsSubstring("\\d+ errors from");
    const auto errosPos = analysisOutput.indexOf(DynamicErrorsSubstring);

    if (errosPos > -1)
    {
        const QRegularExpression CounterSubstring("\\d+");
        dynamicCount = CounterSubstring.match(analysisOutput,
errosPos).captured().toInt();
    }

    static const QRegularExpression CWEErrorSubstring("\\[CWE\\d+\\]");
    auto staticCount = analysisOutput.count(CWEErrorSubstring);

    QString outputString;
    QString stylesheet;

    if (dynamicCount >= 0)
    {
        outputString.append(QString("Dynamic errors: %1.").arg(dynamicCount));
        stylesheet = "color: rgb(230, 150, 25);";
    }

    if (staticCount >= 0)
    {
        outputString.append(QString("Static errors: %1.").arg(staticCount));
        stylesheet = "color: rgb(230, 150, 25);";
    }
```

```
    if (staticCount >= 0 && dynamicCount >= 0 && staticCount != dynamicCount)
    {
        outputString.append("Possible false-positives. Be careful!");
        stylesheet = "color: red;";
    }

    destination->setText(outputString);
    destination->setStyleSheet(stylesheet);
}

QString getElapsedAnalysisTimeString(const int msec = 0)
{
    return QString("Time spent: %1").arg(msec);
}
} // anonymous


class OperationsPool : public QObject
{
Q_OBJECT

public:
    using QObject::QObject;

    void appendOperation(AsyncOperation* operation)
    {
        if (operation == nullptr)
        {
            return;
```

```
        }

        _operations.append(operation);

        connect(operation, &AsyncOperation::finished, this, [this, operation]
        {
            _operations.removeOne(operation);

            if (_operations.isEmpty())
            {
                emit allOperationsFinished();

                deleteLater();
            }
        });
    }

signals:
    void allOperationsFinished();

private:
    QList<AsyncOperation*> _operations;
};

MainWindow::MainWindow(QWidget* parent)
    : QMainWindow(parent)
{
    setupUI();
}
```

```
void MainWindow::setupUI()
{
    QWidget* centralWidget = new QFrame(this);
    setCentralWidget(centralWidget);

    QVBoxLayout* centralLayout = new QVBoxLayout(centralWidget);

    centralLayout->addWidget(createSourcesAnalysisWidgetsGroup());
    centralLayout->addWidget(createBinariesAnalysisWidgetsGroup());
    centralLayout->addWidget(createRunAsyncAnalysisPredicateWidget());
    centralLayout->addWidget(createOutputWidgetsBlock());
    centralLayout->addStretch();
    centralLayout->addWidget(createAnalysisRunningStateWidget());

    setMinimumSize(472, 360);
}

QWidget* MainWindow::createSourcesAnalysisWidgetsGroup()
{
    QWidget* group = new QGroupBox("Static sources analysis", this);
    QHBoxLayout* hLayout = new QHBoxLayout(group);

    _enableStaticSourcesCheckbox = new QCheckBox("Run static sources analysis",
group);
    hLayout->addWidget(_enableStaticSourcesCheckbox);

    _pathToSources = new QLineEdit(group);
    hLayout->addWidget(_pathToSources);

    QPushButton* browsePathButton = new QPushButton("Browse sources", group);
```

```
    hLayout->addWidget(browsePathButton);

    connect(browsePathButton,
            &QPushButton::clicked,
            this,
            [this, browsePathButton]()
            {
                QString dir = QFileDialog::getExistingDirectory(browsePathButton,
                                                                 "Choose directory with cxx sources",

QStandardPaths::writableLocation(QStandardPaths::HomeLocation),
                                                                 QFileDialog::ShowDirsOnly |
                                                                 QFileDialog::DontResolveSymlinks);

                _pathToSources->setText(dir);
            });

    return group;
}

QWidget* MainWindow::createBinariesAnalysisWidgetsGroup()
{
    QWidget* group = new QGroupBox("Binaries (libraries and executables)
analysis", this);
    QGridLayout* grid = new QGridLayout(group);
    _enableStaticBinaryCheckbox = new QCheckBox("Run static binaries analysis",
group);
    grid->addWidget(_enableStaticBinaryCheckbox, 0, 0);
```

```
    _enableDynamicBinaryCheckbox = new QCheckBox("Run dynamic binaries
analysis", group);
    grid->addWidget(_enableDynamicBinaryCheckbox, 1, 0);

    _pathToBinary = new QLineEdit(group);
    grid->addWidget(_pathToBinary, 0, 1);

    QPushButton* browsePathButton = new QPushButton("Browse binaries", group);
    grid->addWidget(browsePathButton, 0, 2);

    connect(browsePathButton,
            &QPushButton::clicked,
            this,
            [this, browsePathButton]()
            {
                QString dir = QFileDialog::getOpenFileName(browsePathButton,
                                                "Choose binary executable",

QStandardPaths::writableLocation(QStandardPaths::HomeLocation));

                _pathToBinary->setText(dir);
            });

    return group;
}

QWidget* MainWindow::createRunAsyncAnalysisPredicateWidget()
{
    _runAsyncCheckbox = new QCheckBox("Run async analysis", this);
```

```
    return _runAsyncCheckbox;
}

QWidget* MainWindow::createOutputWidgetsBlock()
{
    QWidget* blockWidget = new QWidget(this);
    QVBoxLayout* blockLayout = new QVBoxLayout(blockWidget);

    QGroupBox* sourcesOutput = new QGroupBox("Sources analysis result: ",this);
    QVBoxLayout* sourcesOutputLayout = new QVBoxLayout(sourcesOutput);

    _sourcesAnalysisOutputBrowser = new QTextBrowser(sourcesOutput);
    _sourcesAnalysisOutputBrowser-
>setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    _sourcesAnalysisOutputBrowser-
>setWordWrapMode(QTextOption::WrapAtWordBoundaryOrAnywhere);
    sourcesOutputLayout->addWidget(_sourcesAnalysisOutputBrowser);

    blockLayout->addWidget(sourcesOutput, 1);

    QGroupBox* binaryOutput = new QGroupBox("Binary analysis result: ",this);
    QVBoxLayout* binaryOutputLayout = new QVBoxLayout(binaryOutput);

    _binaryAnalysisOutputBrowser = new QTextBrowser(binaryOutput);
    _binaryAnalysisOutputBrowser-
>setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    _binaryAnalysisOutputBrowser-
>setWordWrapMode(QTextOption::WrapAtWordBoundaryOrAnywhere);
    binaryOutputLayout->addWidget(_binaryAnalysisOutputBrowser);
```

```
    blockLayout->addWidget(binaryOutput, 1);

    _timeTakenLabel = new QLabel(blockWidget);
    _timeTakenLabel->setText(getElapsedAnalysisTimeString());
    blockLayout->addWidget(_timeTakenLabel);

    _binaryErrorsSummary = new QLabel(blockWidget);
    _binaryErrorsSummary->setText(" ");
    blockLayout->addWidget(_binaryErrorsSummary);

    return blockWidget;
}

QWidget* MainWindow::createAnalysisRunningStateWidget()
{
    QWidget* widget = new QWidget(this);
    QHBoxLayout* hLayout = new QHBoxLayout(widget);

    _progressBar = new QProgressBar(widget);
    _progressBar->setRange(0, 100);
    constexpr auto AutoIntrementMaxValue = 85;

    hLayout->addWidget(_progressBar);

    QTimer* autoProgressTimer = new QTimer(_progressBar);
    autoProgressTimer->setInterval(500);
    connect(autoProgressTimer,
            &QTimer::timeout,
            _progressBar,
            [this, autoProgressTimer]()
```

```
        {
            constexpr auto SingleStep = 2;
            const auto progressValue = std::min(_progressBar->value() + SingleStep,
_progressBar->maximum());

            _progressBar->setValue(progressValue);

            if (progressValue >= AutoIntrementMaxValue)
            {
                autoProgressTimer->stop();
                autoProgressTimer->deleteLater();
            }
        });

    hLayout->addStretch();

    connect(_progressBar,
        &QProgressBar::valueChanged,
        this,
        [this, autoProgressTimer](const int value)
        {
            if (_progressBar->value() >= AutoIntrementMaxValue)
            {
                autoProgressTimer->stop();
            }
        });

    QPushButton* runButton = new QPushButton("Run", this);
    connect(runButton,
        &QPushButton::clicked,
```

```
        this,
        [this, autoProgressTimer]()
        {
            _sourcesAnalysisOutputBrowser->clear();
            _binaryAnalysisOutputBrowser->clear();

            _progressBar->setValue(0);
            autoProgressTimer->start();

            QSharedPointer<QElapsedTimer> timeTakenTimer(new
QElapsedTimer());
            timeTakenTimer->start();
            auto* pool = performAnalysis(_runAsyncCheckbox->isChecked());

            connect(pool, &OperationsPool::allOperationsFinished, this, [this,
timeTakenTimer]
            {
                _timeTakenLabel-
>setText(getElapsedAnalysisTimeString(timeTakenTimer->elapsed()));

                _progressBar->setValue(_progressBar->maximum());

                extractAnalysisErrorsSummary(_binaryAnalysisOutputBrowser-
>toPlainText(), _binaryErrorsSummary);
            });
        });
    hLayout->addWidget(runButton);

    return widget;
}
```

```
OperationsPool* MainWindow::performAnalysis(const bool runAsync)
{
   auto performAnalysisImpl = [this, runAsync](QCheckBox* predicate,
                                               QTextBrowser* outputContainer,
                                               std::function<QString(std::string)> analysisFunc,
                                               const QString& path,
                                               const QString& emptyPathFallbackMessage) ->
AsyncOperation*
   {
      assert(predicate);
      assert(outputContainer);

      if (path.isEmpty() || !predicate->isChecked())
      {
         if (predicate->isChecked())
         {
            outputContainer->append(emptyPathFallbackMessage);
         }

         return nullptr;
      }

      auto* op = new AsyncOperation([this, func = std::move(analysisFunc), path =
path.toStdString()]()
                                    {
                                       return func(path);
                                    },
                                    runAsync);
```

```
    connect(op,
            &AsyncOperation::finished,
            this,
            [this, outputContainer](QVariant result)
            {
                outputContainer->append(result.toString());
            });

    if (runAsync)
    {
        op->run();
    }

    return op;
};

OperationsPool* pool = new OperationsPool(this);

auto* staticSourcesOp = performAnalysisImpl(

                    _enableStaticSourcesCheckbox,
                    _sourcesAnalysisOutputBrowser,
                    [this](std::string path)
                    {
                        return QString::fromStdString(
                                _caller.callStaticSourcesAnalysis(std::move(path)));
                    },
                    _pathToSources->text(),
                    "No path specified for sources. Sources analysis will be skipped.");
```

```
auto* dynamicBinaryOp = performAnalysisImpl(
                    _enableDynamicBinaryCheckbox,
                    _binaryAnalysisOutputBrowser,
                    [this](std::string path)
                    {
                       return QString::fromStdString(
                             _caller.callDynamicBinaryAnalysis(std::move(path)));
                    },
                    _pathToBinary->text(),
                    "No path specified for binary. Binary analysis will be skipped.");


auto* staticBinaryOp = performAnalysisImpl(
                    _enableStaticBinaryCheckbox,
                    _binaryAnalysisOutputBrowser,
                    [this](std::string path)
                    {
                       return QString::fromStdString(
                             _caller.callStaticBinaryAnalysis(std::move(path)));
                    },
                    _pathToBinary->text(),
                    "No path specified for binary. Binary analysis will be skipped.");

if (runAsync)
{
   pool->appendOperation(staticSourcesOp);
   pool->appendOperation(dynamicBinaryOp);
   pool->appendOperation(staticBinaryOp);

}
```

```
    else
    {
        auto* cumulativeOperation = new AsyncOperation([=]()
                                        {
                                            if (staticSourcesOp)
                                            {
                                                staticSourcesOp->run();
                                            }
                                            if (dynamicBinaryOp)
                                            {
                                                dynamicBinaryOp->run();
                                            }
                                            if (staticBinaryOp)
                                            {
                                                staticBinaryOp->run();
                                            }
                                            return QVariant();
                                        });

        cumulativeOperation->run();

        pool->appendOperation(cumulativeOperation);
    }

    connect(pool, &OperationsPool::allOperationsFinished, pool,
&OperationsPool::deleteLater);

    return pool;
}
```

```
std::string callCWECheckerProcess(const std::string forPath)
{
    QProcess cweCheckerProc;

    QString executable = CWECheckerExecutable;
    QStringList arguments;
    arguments << QString::fromStdString(forPath);

    cweCheckerProc.start(executable, arguments);
    cweCheckerProc.waitForFinished(-1);

    const auto stdOut = cweCheckerProc.readAll().toStdString();
    const auto stdErr = cweCheckerProc.readAllStandardError().toStdString();

    return stdOut + '\n' + stdErr;
}

std::string callCppCheckProcess(const std::string forPath)
{
    QProcess cppCheckProc;

    QString executable = CppCheckExecutable;
    QStringList arguments;
    arguments << QString::fromStdString(forPath) << "--force" << "--inconclusive";

    cppCheckProc.start(executable, arguments);
    cppCheckProc.waitForFinished(-1);

    const auto stdOut = cppCheckProc.readAll().toStdString();
```

```
    const auto stdErr = cppCheckProc.readAllStandardError().toStdString();

    return stdOut + '\n' + stdErr;
}

std::string callValgrindCheckProcess(const std::string forPath)
{
    QProcess valgrindCheckProc;

    QString executable = ValgrindExecutable;
    QStringList arguments;
    arguments << QString::fromStdString(forPath);

    valgrindCheckProc.start(executable, arguments);
    valgrindCheckProc.waitForFinished(-1);

    const auto stdOut = valgrindCheckProc.readAll().toStdString();
    const auto stdErr = valgrindCheckProc.readAllStandardError().toStdString();

    return stdOut + '\n' + stdErr;
}

class AnalysisCaller {
public:
    std::string callStaticSourcesAnalysis(std::string forPath);
    std::string callStaticBinaryAnalysis(std::string forExecutable);
    std::string callDynamicBinaryAnalysis(std::string forPath);
};
std::string AnalysisCaller::callStaticSourcesAnalysis(std::string forPath)
{
```

```
    return callCppCheckProcess(forPath);
}

std::string AnalysisCaller::callStaticBinaryAnalysis(std::string forExecutable)
{
    return callCWECheckerProcess(forExecutable);
}

std::string AnalysisCaller::callDynamicBinaryAnalysis(std::string forPath)
{
    return callValgrindCheckProcess(forPath);
}
```

# ДОДАТОК В

## Лістинги скриптів конфігурації компоненту ExternalDependencies

```
set(EXTERNAL_DEPENDENCIES_DIR_NAME
"MixedAnalysisExternalDependencies")
set(EXTERNAL_DEPENDENCIES_PATH
"$ENV{HOME}/${EXTERNAL_DEPENDENCIES_DIR_NAME}")
message(STATUS "Setting up project dependencies. Working dir:
${EXTERNAL_DEPENDENCIES_PATH}")
file(MAKE_DIRECTORY ${EXTERNAL_DEPENDENCIES_PATH})
add_custom_target(${EXTERNAL_DEPENDENCIES_SRC_DIR})

macro(CHECK_EXTERNAL_COMPONENT NAME FOUND_OUT_VAR)
   file(GLOB FILES "${CMAKE_BINARY_DIR}/${NAME}/*")
   list(LENGTH FILES FILES_COUNT)
   if(${FILES_COUNT} GREATER 0)
      set("${FOUND_OUT_VAR}" TRUE)
   else()
      set("${FOUND_OUT_VAR}" FALSE)
   endif()
endmacro()

macro(SETUP_EXTERNAL_EXECUTABLE_COMPONENT NAME
EXECUTABLE_NAME EXECUTABLE_PATH_OUT)
   message(STATUS "Processing ${NAME}")
   set(ALREADY_AVAILABLE)
   CHECK_EXTERNAL_COMPONENT("${NAME}" ALREADY_AVAILABLE)
   if (${ALREADY_AVAILABLE})
      message(STATUS "Found non-empty ${NAME}. Setup step skipped.")
```

```
    else()
        add_subdirectory("${EXTERNAL_DEPENDENCIES_SRC_DIR}/${NAME}")
        add_dependencies(${EXTERNAL_DEPENDENCIES_SRC_DIR} ${NAME})
    endif()

find_program("${EXECUTABLE_PATH_OUT}" "${EXECUTABLE_NAME}"
REQUIRED HINTS "${CMAKE_BINARY_DIR}/${NAME}/*")
    message(STATUS "Found ${NAME} executable at:
${${EXECUTABLE_PATH_OUT}}")
endmacro()

set(CWE_CHECKER_COMPONENT_NAME "CweChecker")
SETUP_EXTERNAL_EXECUTABLE_COMPONENT(${CWE_CHECKER_COM
PONENT_NAME} "cwe_checker" CWE_CHECKER_EXECUTABLE)
set(CPP_CHECK_COMPONENT_NAME "CppCheck")
SETUP_EXTERNAL_EXECUTABLE_COMPONENT(${CPP_CHECK_COMPON
ENT_NAME} "cppcheck" CPP_CHECK_EXECUTABLE)
set(VALGRIND_COMPONENT_NAME "Valgrind")
SETUP_EXTERNAL_EXECUTABLE_COMPONENT(${VALGRIND_COMPON
ENT_NAME} "valgrind" VALGRIND_EXECUTABLE)
set(QT_COMPONENT_NAME "Qt5")
# NOTE: Use include instead of add_subdirectory to make Qt packages visible in the
parent score.
include(${EXTERNAL_DEPENDENCIES_SRC_DIR}/${QT_COMPONENT_NA
ME}/CMakeLists.txt)

add_custom_command(TARGET ${EXTERNAL_DEPENDENCIES_SRC_DIR}
POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E rm -rf
${EXTERNAL_DEPENDENCIES_PATH})
```

```
include(ExternalProject)

add_custom_target(${VALGRIND_COMPONENT_NAME})

set(VALGRIND_SRC_DIR "valgrind")
set(VALGRIND_HOME_PATH
"${EXTERNAL_DEPENDENCIES_PATH}/${VALGRIND_COMPONENT_NAM
E}")
set(VALGRIND_SRC_PATH
"${EXTERNAL_DEPENDENCIES_PATH}/${VALGRIND_SRC_DIR}")
set(VALGRIND_BUILD_PATH
"${CMAKE_BINARY_DIR}/${VALGRIND_COMPONENT_NAME}")

configure_file("ExternalCMakeProject/CMakeLists.txt.in"
"ExternalCMakeProject/CMakeLists.txt")
set(EXTERNAL_CMAKE_PROJECT_DIR "ExternalCMakeProject")
set(EXTERNAL_CMAKE_PROJECT_WORKING_DIR
"${CMAKE_CURRENT_BINARY_DIR}/${EXTERNAL_CMAKE_PROJECT_DI
R}")
execute_process(COMMAND "${CMAKE_COMMAND}" -G
"${CMAKE_GENERATOR}"
    "${EXTERNAL_CMAKE_PROJECT_WORKING_DIR}"
WORKING_DIRECTORY
"${EXTERNAL_CMAKE_PROJECT_WORKING_DIR}"
    )

execute_process(COMMAND "${CMAKE_COMMAND}" --build
    "${EXTERNAL_CMAKE_PROJECT_WORKING_DIR}"
WORKING_DIRECTORY
"${EXTERNAL_CMAKE_PROJECT_WORKING_DIR}"
```

```
    )

cmake_minimum_required(VERSION 3.15)
include(ExternalProject)

project("${VALGRIND_COMPONENT_NAME}_Setup_Project" NONE)

ExternalProject_Add(
    ${VALGRIND_COMPONENT_NAME}
    PREFIX                  "${VALGRIND_HOME_PATH}"
    TMP_DIR                 "${VALGRIND_HOME_PATH}/tmp"
    STAMP_DIR               "${VALGRIND_HOME_PATH}/stamp"
    DOWNLOAD_DIR            "${VALGRIND_SRC_PATH}"
    SOURCE_DIR              "${VALGRIND_SRC_PATH}"
    GIT_REPOSITORY          "git://sourceware.org/git/valgrind.git"
    GIT_TAG                 "VALGRIND_3_18_1"
    CONFIGURE_COMMAND       ./autogen.sh && ./configure --
prefix=${VALGRIND_BUILD_PATH}
    BUILD_COMMAND           make -j4
    BUILD_IN_SOURCE         1
    INSTALL_COMMAND         make install
)


include(ExternalProject)

if (NOT DEFINED PATH_TO_GHIDRA)
   message(FATAL_ERROR "No ghidra path specified for cwe checker.")
endif()
```

```
set(CWE_CHECKER_SRC_DIR "cwe_checker_lib")
set(CWE_CHECKER_HOME_PATH
"${EXTERNAL_DEPENDENCIES_PATH}/${CWE_CHECKER_COMPONENT_
NAME}")
set(CWE_CHECKER_SRC_PATH
"${EXTERNAL_DEPENDENCIES_PATH}/${CWE_CHECKER_SRC_DIR}")
set(CWE_CHECKER_BUILD_PATH
"${CMAKE_BINARY_DIR}/${CWE_CHECKER_COMPONENT_NAME}")
set(PATCH_FILE
"${CMAKE_CURRENT_SOURCE_DIR}/Patches/cweCheckerPatch.diff")

configure_file("ExternalCMakeProject/CMakeLists.txt.in"
"ExternalCMakeProject/CMakeLists.txt")
set(EXTERNAL_CMAKE_PROJECT_DIR "ExternalCMakeProject")
set(EXTERNAL_CMAKE_PROJECT_WORKING_DIR
"${CMAKE_CURRENT_BINARY_DIR}/${EXTERNAL_CMAKE_PROJECT_DI
R}")
execute_process(COMMAND "${CMAKE_COMMAND}" -G
"${CMAKE_GENERATOR}"
    "${EXTERNAL_CMAKE_PROJECT_WORKING_DIR}"
WORKING_DIRECTORY
"${EXTERNAL_CMAKE_PROJECT_WORKING_DIR}"
    )

execute_process(COMMAND "${CMAKE_COMMAND}" --build
    "${EXTERNAL_CMAKE_PROJECT_WORKING_DIR}"
WORKING_DIRECTORY
"${EXTERNAL_CMAKE_PROJECT_WORKING_DIR}"
    )
```

```
cmake_minimum_required(VERSION 3.15)
include(ExternalProject)

project("${CWE_CHECKER_COMPONENT_NAME}_Setup_Project" NONE)

ExternalProject_Add(
    CweCheckerSetupLib
    PREFIX                  "${CWE_CHECKER_HOME_PATH}"
    TMP_DIR                 "${CWE_CHECKER_HOME_PATH}/tmp"
    STAMP_DIR               "${CWE_CHECKER_HOME_PATH}/stamp"
    DOWNLOAD_DIR
"${CWE_CHECKER_HOME_PATH}/${CWE_CHECKER_SRC_DIR}"
    GIT_REPOSITORY          "https://github.com/fkie-cad/cwe_checker"
    GIT_TAG                 "master"
    CONFIGURE_COMMAND       ""
    SOURCE_DIR
"${CWE_CHECKER_HOME_PATH}/${CWE_CHECKER_SRC_DIR}"
    BINARY_DIR
"${CWE_CHECKER_HOME_PATH}/${CWE_CHECKER_SRC_DIR}"
    PATCH_COMMAND           git apply ${PATCH_FILE}
    BUILD_COMMAND           make all
GHIDRA_PATH=${PATH_TO_GHIDRA}
TARGET_DIR=${CWE_CHECKER_BUILD_PATH}
    INSTALL_COMMAND         ""
    LOG_CONFIGURE           1
    LOG_BUILD               0
    LOG_INSTALL             1
    LOG_TEST                0)

include(ExternalProject)
```

```
add_custom_target(${CPP_CHECK_COMPONENT_NAME})

set(CPP_CHECK_SRC_DIR "cppcheck_lib")
set(CPP_CHECK_HOME_PATH
"${EXTERNAL_DEPENDENCIES_PATH}/${CPP_CHECK_COMPONENT_NA
ME}")
set(CPP_CHECK_SRC_PATH
"${CPP_CHECK_HOME_PATH}/${CPP_CHECK_SRC_DIR}")
set(CPP_CHECK_BUILD_PATH
"${CMAKE_BINARY_DIR}/${CPP_CHECK_COMPONENT_NAME}")

# Dilemma:
# 1) An external executable can be found during the root project configuration, in
which case we will know its location
#    at the configuration and compilation respectively. But it looks a bit hacky.
# 2) An external executable can be found at runtime.  It seems that this option is not
better...
configure_file("ExternalCMakeProject/CMakeLists.txt.in"
"ExternalCMakeProject/CMakeLists.txt")
set(EXTERNAL_CMAKE_PROJECT_DIR "ExternalCMakeProject")
set(EXTERNAL_CMAKE_PROJECT_WORKING_DIR
"${CMAKE_CURRENT_BINARY_DIR}/${EXTERNAL_CMAKE_PROJECT_DI
R}")
execute_process(COMMAND "${CMAKE_COMMAND}" -G
"${CMAKE_GENERATOR}"
      "${EXTERNAL_CMAKE_PROJECT_WORKING_DIR}"
WORKING_DIRECTORY
"${EXTERNAL_CMAKE_PROJECT_WORKING_DIR}"
)
```

```
execute_process(COMMAND "${CMAKE_COMMAND}" --build
      "${EXTERNAL_CMAKE_PROJECT_WORKING_DIR}"
WORKING_DIRECTORY
"${EXTERNAL_CMAKE_PROJECT_WORKING_DIR}"
)

cmake_minimum_required(VERSION 3.15)
include(ExternalProject)

project("${CPP_CHECK_COMPONENT_NAME}_Setup_Project" NONE)

ExternalProject_Add("${CPP_CHECK_COMPONENT_NAME}_Setup"
   PREFIX                      "${CPP_CHECK_HOME_PATH}"
   TMP_DIR                     "${CPP_CHECK_HOME_PATH}/tmp"
   STAMP_DIR                   "${CPP_CHECK_HOME_PATH}/stamp"
   DOWNLOAD_DIR                "${CPP_CHECK_SRC_PATH}"
   SOURCE_DIR                  "${CPP_CHECK_SRC_PATH}"
   GIT_REPOSITORY              "https://github.com/danmar/cppcheck"
   GIT_TAG                     "main"
   CONFIGURE_COMMAND           cmake ${CPP_CHECK_SRC_PATH} -
DUSE_MATCHCOMPILER=ON
   BINARY_DIR                  "${CPP_CHECK_BUILD_PATH}"
   BUILD_COMMAND               cmake --build ./
   BUILD_IN_SOURCE             0
   INSTALL_COMMAND             ""
)
```